FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Bioinformatics index tool for elastic degenerate string matching

Bc. Dominika Draesslerová

Department of theoretical computer science
Supervisor: prof. Ing. Jan Holub, Ph.D.

June 21, 2022

Acknowledgements

First of all, I would like to express many thanks to my supervisor prof. Ing. Jan Holub, Ph.D. for the time and extensive guidance through the stringology field and the opportunity to work on this topic. This thesis allowed me to work on the border between algorithmization and biology and allowed me to improve my abilities in the theoretical background hidden behind huge bioinformatics tools. Many appreciations goes to Mgr. Jan Pačes, Ph.D. from CTU Prague and Mgr. Petr Daněček, Ph.D. from CU, who provided me with consultations in the bioinformatics field and introduced me to the major questions and problems of today's genomics. Many thanks to Nicola Gigante and Nicola Prezza from the University of Udine who complied with my request for commissioning of their application and library. Finally, I would like to thank my family and friends for the big support not only in writing this work but during my whole studies at the university.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a non-exclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on June 21, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Draesslerová, Dominika. *Bioinformatics index tool for elastic degenerate string matching.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022. Also available from: ⟨`https://github.com/draessld/bio-fmi`⟩.

Abstrakt

Schopnost efektivně prohledávat veliké množství dat je důležitou součástí nejen v bioinformatické a informatické sféře, ale v celém moderním světe. Každý organismus je originální a vytvořit komplexní strukturu, která by vhodně reprezentovala genomy a jeho varianty a uměla s nimi efektivně pracovat, je hlavním směrem pangenomického výzkumu. V této práci diskutujeme a implementujeme poměrně nový algoritmus zvaný BIO-FMI, který má potenciál k tomu efektivně komprimovat a vyhledávat nad množinou vysoce repetitivních dat, jako jsou právě DNA sequence. V současné době je způsob ukládání nevyhovující vzhledem k postavení vůči jednomu referenčnímu genomu a hledají se alternativní řešení. Tato práce se zabývá modifikací algoritmu BIO-FMI na formát elastických degenerovaných řetězců (EDS), které jsou kandidátními reprezentanty pro ukládání variant. Práce ukazuje slibné výsledky v rychlosti sestavení indexu, variabilitě nastavení a porovnává je s dalšími algoritmy z této oblasti, kterými jsou LZ-RLBWT a r-index.

Klíčová slova BIO-FMI, elastický degenerovaný řetězce, self-index, pangenomatické prohledávání

Abstract

The ability to search efficiently over large amount of data is an important part not only the field of bioinformatics but throughout the modern time. Every organism is unique, and to create a complex structure appropriately representing genomes and their variants while at the same time being able to work with them efficiently seems to be the main pan-genomic research direction. This text deals with the discussions about the implementation of a relatively new algorithm called BIO-FMI with the potential to efficiently compress and search over a set of highly repetitive strings, such as the DNA sequences. The storage principles of variants are currently insufficient due to their position against one reference genome and some alternative solutions are being sought now. This thesis specializes in a modification of algorithm BIO-FMI to the format of elastic-degenerate strings which become candidate representatives in terms of storing variants. The thesis shows promising results in index construction time, setting variability while comparing them with other algorithms in this field, namely LZ-RLBWT and r-index.

Keywords BIO-FMI, elastic-degenerate strings, self-index, pangenomatic pattern matching

Contents

List of Figures

List of Tables

Introduction

Motivation and objectives

Large genomics data collections and biological computation are expanding departments these days. Biological data such as DNA and RNA sequences contain a lot of information, which is needed to store and prepare for queries. In the human genome case, the one chromosome has more than 3×10^9 base pare, which is about 250 Mb of information per one chromosome. Queering over data of this kind consumes a lot of memory and computation time, but it is necessary for bioinformatics analysis. The information from genome sequencing is insignificant without the next preparation and preprocessing. The sequence of characters needs to go through statistical evaluation, mapping in other genomes, or assembly processing to store this information into biological databases and perform the next workflows.

Indexing and compressing formats that store this information are almost necessary because every query on the existence or positions of genes, sub-genes, repetitions, etc. would be over-consumed without it. This document describes the basic problems and notation of indexing and compression algorithms used on a set of genetic information. This is useful mainly for pan-genomic and variant studies, where a lot of genomes are compared to each other, and a set of genomes is the main target of studies. In short, a set of genomes compares the genetic equipment of familiar organisms or the same kind of organisms in one complex structure.

This document will describe a relatively new indexing method over a set of strings called BIO-FMI. It has the potential to be a prospective index of bioinformatics formats, like variant calling format (VCF), and according to this, it could help with querying a huge amount of information flowing from genome variant data. Next, we will design an index over elastic degenerated strings (EDS) and discuss their corresponding storage structure. EDS structure has the potential to be used as a pan-genomic visualization and saving format.

Problem statements

An indexation and compression are closely linked. Biological sequences are often highly repetitive (about 10 % of variability in human genomes) and easily compressed, so the search for short patterns over compressed data is an advantage. The methods when we need to search over a full original text, are known as full-text compression and full-text indexation. When the index works with original text and transforms it to get the best information from that, then we talk about self-indexing.

Searching over the text can be divided into exact and approximate matching, depending on allowed mismatches. A distance between two strings could be defined in many ways according to the used algorithm (Levenshtein distance [5], rating matrixes [43, 33], ...).

It is possible to study genetic variants using assembly techniques over many existing genomes, but in the case of larger genomes, such as the human genome, this method is really time-consuming. Another way is to map our sequence to the existing model organism, or reference genome. Finally, if no model organism is available, denovo assembly is another method that can be used, but the result quality decreases.

Whatever method we use, the results are subject to error (bias), which can lead to misleading observations. The ideal model for this study is a pangenomic picture that gives a general overview of all known variants. Then alignment or insert of new sequence means comparing with the most familiar structure, resp. combination of variants.

The use of pattern matching in genomes is mainly in gene and variant research. It can be used in deep learning to learn the natural language of DNA [28] and pattern mining to identify repetitive sequences that are difficult to read [15].

State of the Art

Indexes used in these days can be divided into two groups with different approaches — non-reference-based indexes (usually based on dictionaries) and reference-based indexes (mostly based on FM-index and BWT). Reference-based genomics searching is popular because the concept of the reference sequence is commonly used. The reference sequence in genomics and proteomics is just a kind of agreed consensus string and because of it, a better way is sought after.

The common format for storing variants is Variant Calling Format (VCF) [34] and the most used tool for pangenome representation is the VG toolkit [18]. The tool is based on the structure of variant graphs, which is very complex due to their tree structure.

In 2015, a collective of researchers from Finland, Italy and France came up with the proposition of index based on a combination of Lempel Ziv dictionary compression algorithm with run-length Burrows-Wheeler transformation, which gives a firm base for a new generation of genome indexing. The improvement called the r-index [17] published in 2017 is currently probably the most examined tool for this purpose.

First, in this text we will define basic notions and describe the required algorithms. Afterward, the exact matching algorithm called BIO-FMI will be introduced in detail and further improved to accept an elastic degenerate string as an input type. Time and memory complexity will be described for both types. Then we will focus on the implementation and experiment details and finally, our results will be discussed.

Chapter 1

Basic notions

It is necessary to define several related stringology terms to clearly explain the basic principles of our algorithms. Furthermore, there is great interest in explaining the algorithms that are indispensable parts of our tool. Basic stringology definitions are taken from stringology lectures at my university [22] and the elastic degenerate string notation is inspired by Costas S. Iliopoulos et al. [23].

1.1 Basic definitions

Definition 1.1.1 (Alphabet). *An alphabet Σ is a finite non-empty ordered set of symbols. We will refer to Σ as a set of nucleotide symbols A, C, G, T, N, expanded by characters $-, \#$, and \$.*

Definition 1.1.2 (String). *String x over an alphabet Σ is a concatenation of alphabet symbols. The size of string $x = x[1]x[2]\dots x[n]$ is the number of symbols in the string. We denote it as $|x| = n$.*

Definition 1.1.3 (Substring). *The substring of string x is $x[i..j] = x[i]x[i+1]\dots x[j], 1 \leq i \leq j \leq |x| = n$. If $i = 1$, then $x[i..j]$ is called prefix of x. If $j = n$, than $x[i..j]$ is called suffix of x. If $i > j$, then $x[i..j] = \varepsilon$.*

Definition 1.1.4 (K-mer). *A k-mer u is a substring of string T of length k.*

Definition 1.1.5 (Rank operation). *The $rank_b(B, i)$ operation is defined as the number of occurrences of bit b in $B[1..i]$, where $b \in \{0, 1\}$ and B is a bit vector of length $n \geq i$.*

Definition 1.1.6 (Select operation). *The $select_b(B, i)$ operation is defined as the position of i-th occurrence of symbol b in B, where $b \in \{0, 1\}$ and B is a bit vector of length $n \geq i$.*

Definition 1.1.7 (Elastic-degenerate symbol)**.** *An elastic-degenerate symbol ξ, over a given alphabet Σ is a set of two or more strings over Σ. We denote it as* $\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_k \end{bmatrix}, k = |\xi|$. *Also we define minimum (resp. maximum) length in ξ as the length of shortest (resp. longest) string $E_i \in \xi$, denote by $|\xi|_{min}$ (resp. $|\xi|_{max}$).*

Definition 1.1.8 (Elastic-degenerate string (EDS))**.** *An elastic-degenerate string $\hat{x}$ over alphabet Σ is a sequence $s_1\xi_1 s_2\xi_2 \dots s_{k-1}\xi_{k-1}s_k$, where each $s_i \in \Sigma^* \setminus \{\varepsilon\}, 0 \leq i \leq k$ is an string called* ***seed*** *and each $\xi_i, 0 \leq i \leq k-1$ is a elastic-degenerate symbol.*

Definition 1.1.9 (Total size of $\hat{X}$)**.** *The total size of $\hat{X}$, denoted by $||\hat{X}||$, is defined as the sum of the total length of its seeds and the total length of all the strings in each of its elastic-degenerate symbols: $||\hat{X}|| = \sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|$.*

Definition 1.1.10 (Minimum length of $\hat{X}$)**.** *The length of $\hat{X}$, denoted by $|\hat{X}|$, is defined as the sum of the total length of its seeds and minimum length of each elastic-degenerate symbol: $\sum_{i=1}^{k} |S_i| + |\xi_i|_{min}$.*

A position in EDS is given by its minimal length when the first sequence of each symbol is used or is moved about the corresponding offset when the sequence in the symbol is different.

Example 1.1.1. $\hat{x} = \texttt{atg}\begin{bmatrix} \varepsilon \\ \texttt{tt} \end{bmatrix}\texttt{gtc}\begin{bmatrix} \texttt{a} \\ \texttt{cc} \\ \texttt{tga} \end{bmatrix}\texttt{aa}$, *where*

- $s_1 = \texttt{atg}$, $s_2 = \texttt{gtc}$, $s_3 = \texttt{aa}$ *are the seeds of $\hat{x}$,*
- $\xi_1 = \begin{bmatrix} \varepsilon \\ tt \end{bmatrix}$, $\xi_2 = \begin{bmatrix} a \\ cc \\ tga \end{bmatrix}$ *are elastic-degenerate symbols.*
- *For ξ_1: $E_1 = \varepsilon$, $E_2 = tt$, minimal length $|\xi|_{min} = 0$ and maximal length $|\xi|_{max} = 2$*
- *For ξ_2: $E_1 = a$, $E_2 = cc$, $E_3 = tga$, minimal length $|\xi|_{min} = 1$ and maximal length $|\xi|_{max} = 3$*

In the following section, we briefly introduce all the fundamental structures and algorithms that give the solid basis for our tool.

1.2 Suffix array

A suffix array (SA) for string T is a structure of lexicographically sorted suffixes designed in [29]. On position i, the suffix array contains a pointer to the i-th smallest suffix of the original string, that means for any $i, j \in 1 \dots n; i < j : T[SA[i]] < T[SA[j]]$.

There is an algorithm that can search for a pattern P in the original text T using the suffix array in $\mathcal{O}(|P| + \log(|T|))$ time.

Example 1.2.1 (Suffix array construction)**.** *Let text* $T =$ `abraca`

Suffixes of text T with start positions:

```
0: abraca$
1: braca$
2: raca$
3: aca$
4: ca$
5: a$
6: $
```

Sorted suffixes:

```
6: $
5: a$
0: abraca$
3: aca$
1: braca$
4: ca$
2: raca$
```

⇒

i	0	1	2	3	4	5	6
$T[i]$	a	b	r	a	c	a	$
$SA[i]$	6	1	4	2	5	3	0

SA is constructed by the lexicographic ordering of the whole text suffixes as shown in Example 1.2.1. The original principle of SA should store the entire original text. For long texts such as DNA sequences, it could be quite inefficient and therefore has led to further research in the field of compression and sampling over SA [21, 2].

1.3 Burrows-Wheeler transform

The Burrows-Wheeler method was designed by M. Burrow and D. J. Wheeler in [7] as a lossless data compression method. However, only the first part of the algorithm is relevant for our project due to its content organization. The principle of transformation is to create all cyclic shifts of the original string and to sort them lexicographically. It could be visualized as a matrix M of size $n \times n$, where n is the length of the original string. The result of Burrows-Wheeler transformation (BWT) is the last column of the matrix M. We denote the last column as L and the first column as F.

An example of BWT construction over string `abraca` is in Example 1.3.1.

Example 1.3.1 (BWT construction)**.**

1	2	3	4	5	6	7
a	b	r	a	c	a	$
b	r	a	c	a	$	a
r	a	c	a	$	a	b
a	c	a	$	a	b	r
c	a	$	a	b	r	a
a	$	a	b	r	a	c
$	a	b	r	a	c	a

⇒

F						L
1	2	3	4	5	6	7
$	a	b	r	a	c	a
a	$	a	b	r	a	c
a	b	r	a	c	a	$
a	c	a	$	a	b	r
b	r	a	c	a	$	a
c	a	$	a	b	r	a
r	a	c	a	$	a	b

Table 1.1: Burrows-Wheeler transformation simulation on string x=abraca

One thing to note is that we do not have to store the whole matrix M. Because of the transformation characteristics, columns F and L are sufficient. The BWT could be constructed using a suffix array with a formula in Equation 1.1. More are shown later in FM-index section.

$$BWT[i] = \begin{cases} T_{SA[i]-1}, & SA[i] > 1, \\ T_n, & SA[i] = 1. \end{cases} \tag{1.1}$$

The original string is easily reconstructed from the transformed L string with an inverse run of the algorithm. It can be observed that any column of the matrix M is a permutation of the original string T. Thus, we start with our L column and, with every next iteration of the algorithm, we sort all rows lexicographically and vertically concatenated with L from the left to all rows. After n iterations, the whole matrix M is reconstructed.

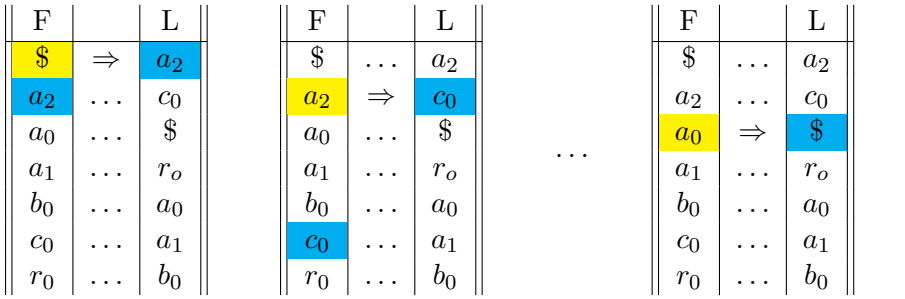
Another reconstruction can be done using the last to front mapping (LF mapping) which has the following property.

Lemma 1.3.1 (LF mapping property)**.** *Let's L is the last column in the BTW matrix and the F is the first column in the BTW matrix. Then the LF mapping property says that the i-th occurrence of a character $c \in \Sigma$ in L corresponds to the i-th occurrence of a character $c \in \Sigma$ in F [26].*

The L and F strings are required for the reconstruction of T. The order of each character is preserved. We know, that the symbol `$` means the end of the whole text T. We also know it is the right context of the previous character. The process is subscribed in Lemma 1.3.1 and its example in Example 1.3.2. Because the reconstruction is done from the last symbols in the T, it is called the backward-search approach.

Example 1.3.2 (LF mapping)**.**
$T = a_0 b_0 r_0 a_1 c_0 a_2 \$$

BWT itself is not an index, but it facilitates editing the original text to a suitable form for the next indexing structures. This is the reason why a lot of compression and indexing techniques work with this kind of transformation.



1.4 Rank and Select query support in constant time

Rank and select structures in constant time query support were proposed by Jacobson, Munro, and Clark [30, 24, 9]. The implementation takes $n + \mathcal{O}(n)$ bits, where n is the length of the bit vector and they are known as Jacobson's rank and Clark's select. This time he showed that attaching a dictionary of size $o(n)$ to the bit vector $B_{1,n}$ is sufficient to support rank operation in constant time on the RAM model. It splits the vector into chunks of $\log^2 n$ bits each and store pre-calculated a cumulative rank up to each chunk. Then a rank can be decomposed as finding what chunk it is in, looking up cumulative rank, and summing it up with a relative rank within the chunk. A relative chunk is calculated by dividing it into sub-chunks, calculating the cumulative rank of these sub-chunks, and finally adding a relative rank of a specific sub-chunk that could be found in constant time.

To summarize the process of select support in constant time, the structure, as well as a rank, is at first divided into chunks, where each contains $\log^2 n$ 1–bits. After that, it checks each chunk if it has a sparse (longer chunks) or dense (shorter chunks) characteristic. For the sparse chunks, it is easy to get the required query result, but for dense ones, it is again divided into sub-chunk and the whole process is repeated.

Both structures and even more are described in detail by G. Navarro et al. in [32].

1.5 Wavelet tree

The principle of a wavelet tree is to partition the alphabet into two sub-alphabets, to assign them one and zero, and transform the original string into a corresponding bit vector. We separate characters based on ones and zeros into a new substrings and repeat the process again until the alphabet is not of the size one. Finally, we get a tree, where the root is the bit vector of the original string and leaves are the numbers of characters counted. Figure 1.1

depicts the example of the structure of a wavelet tree over string *abracadabra*.

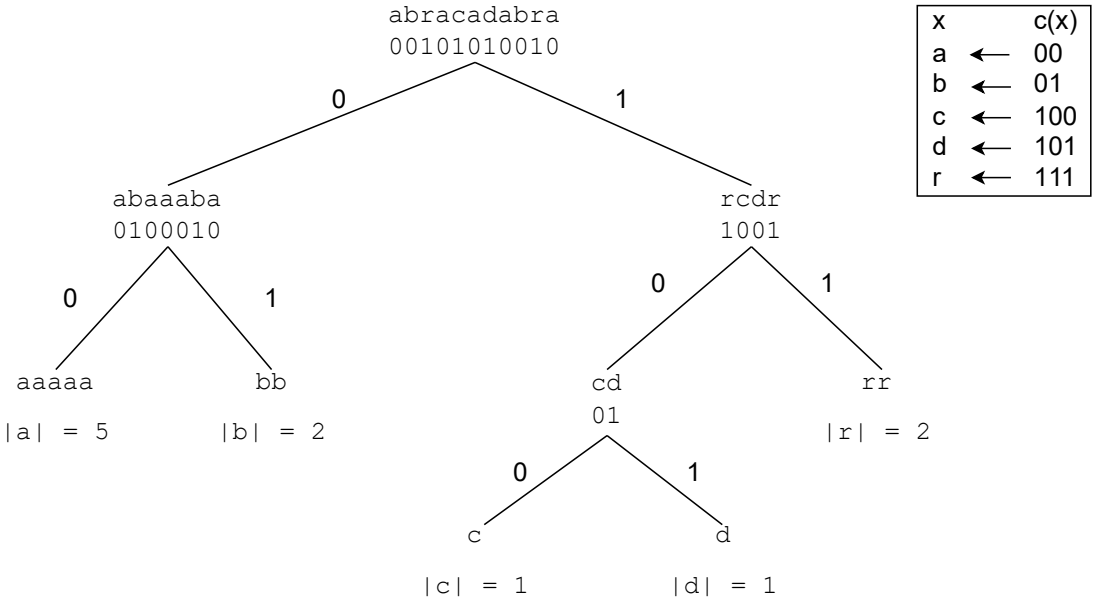


Figure 1.1: Example of a wavelet tree over the string *abracadabra*

When the wavelet tree is balanced, the depth of the tree is $\log |\Sigma|$ and when the rank and select query takes $\mathcal{O}(1)$ time, the time to find one character in $\mathcal{O}(\log n)$ time. Algorithm 1 and Algorithm 2 show how to obtain the required rank and select queries iteratively.

Algorithm 1: $rank_x(i)$

Input : Wavelet tree wt, asked number i, character x
Output: $rank_x(i)$

Function `Search`$((wt, i, x))$`:`
  $N \leftarrow wt.root$;
  $k \leftarrow 0$;
  while N *is not leaf* **do**
    $B \leftarrow N.bitvector$;
    $b \leftarrow c(x)[k]$ `/* get the corresponding bit for character on` k`-th position */`
    $N \leftarrow N.child(b)$;
    $i \leftarrow B.rank_b(i)$;
    $k \leftarrow k + 1$;
  end
  return i;

Algorithm 2: $select_x(i)$

Input : Wavelet tree wt, asked number i, character x
Output: $select_x(i)$

```
1  Function Search((wt, i, x)):
2      N ← wt.leaf(x);
3      l ← |c(x)| − 1;
4      while N is not root do
5          N ← N.parent;
6          B ← N.bitvector;
7          b ← c(x)[k] /* get the corresponding bit for character
               on k-th position                                  */
8          i ← B.select_b(i);
9          k ← k − 1;
10     end
11     return i;
```

1.6 Wavelet tree *FM-index*

Wavelet tree FM-index is a self-index constructed by P. Ferragina and G. Manzini in 2000 [14]. It is a data structure to determine the occurrences of a pattern P in a large text T in a time shorter than $\mathcal{O}(|T| * |P|)$. Its structure is based on backward searching using LF mapping of BWT. The algorithm takes advantage of the linkage of L of F string and more, a count of characters and pointers to their locations can be found using rank queries to the start and end position of the character in $\mathcal{O}(\log \sigma)$ time using the wavelet tree.

Due to relativity between the original text, SA, L, and F string as shown in Table 1.2, there is no need to store the F string.

	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	r	a	c	a	d	a	b	r	a
$SA[i]$	11	8	1	4	6	9	2	5	7	10	3
$SA[i]-1$	10	8	11	3	5	8	1	4	6	9	2
$L = T[SA[i]]$	r	d	a	r	c	a	a	a	a	b	b
$F = T[SA[i]-1]$	a	a	a	a	a	b	b	c	d	r	r

Table 1.2: Structural characteristics and relativity of SA and FM-index

To simulate this algorithm, let us define an additive structure C as an array containing the number of character counts and function $Occ(x, i)$ that returns the number of occurrences of the symbol x in the prefix $L[0 \dots i]$. Each step is shown by Algorithm 3, where in every iteration, the interval $[sp, ep]$ determines the range of occurrences of suffix $P_{i,m}$ in the input text $T_{1,n}$.

Algorithm 3: $FM - search(P, m, n, C, Occ)$

Input : pattern P of length m, length of n, array C containing number of character counts, function Occ
Output: Interval of pattern locations

```
1  Function Search((P, m, n, C, Occ)):
2      sp ← 1;
3      ep ← n;
4      for i ← m to 1 do
5          sp ← C(P_i) + Occ(P_i, sp − 1) + 1;
6          ep ← C(P_i) + Occ(P_i, ep);
7          if sp > ep then
8              return ∅;
9          end
10         i ← i − 1;
11     end
12     return [sp, ep];
```

An array C takes $|\Sigma| \log n$ bits, and function Occ is performed in $\mathcal{O}(\log |\Sigma|)$ time according to the wavelet tree structure. Finally, the counting number of occurrences of $ep - sp + 1$ in the range $[sp, ep]$ takes $\mathcal{O}(|P| \log |\Sigma|)$ time.

1.7 Introduction to genomics notions

In this thesis the algorithms related to genomics are being described and the basic notions from biochemistry including genetics are being defined. The following notions are taken from Bioinformatics and functional genomics by Pevsner, Jonathan [37].

Definition 1.7.1 (Genome)**.** *Genome is all the genetic material in the chromosomes of a particular organism. Its size is generally given as its total number of base pairs.*

Definition 1.7.2 (Gene)**.** *Gene is the fundamental physical and functional unit of heredity. It is an ordered sequence of nucleotides located in a particular position on a particular chromosome that encodes a specific functional product (i.e., a protein or RNA molecule).*

Definition 1.7.3 (Chromosome)**.** *A chromosome is the self-replicating genetic structure of cells containing the cellular DNA that bears in its nucleotide sequence the linear array of genes. In prokaryotes, chromosomal DNA is circular and the entire genome is carried on one chromosome. The eukaryotic genome consists of a number of chromosomes whose DNA is associated with different kinds of proteins.*

1.8 Sequence analysis

Modern methods of sequencing have evolved considerably over the last twenty years. These next-generation sequencing (NGS) methods use parallelization and produce a huge amount of short sequences. In general, there is no possibility to sequence too long samples because of their decreasing quality. In the last years the method of using cell nanopores has gained popularity because of its high quality reading of longer sequences, but these methods stay quite expensive. The most used method to obtain proper quality from sequencing is high-throughput sequencing (HTS) [42], which accumulates a huge amount of reads and using statistical and bioinformatics methods gets the best result. Reads are saved in corresponding bioinformatics format together with their quality for example format SAM [27].

After sequencing, there are many possibilities of what to do depending on the required results and available resources. If there is a model organism and it is known which kind of organism we sequence, then reads are aligned to the reference sequence of this organism and we get a specific position to this genome. Sometimes, the model organism is not available. Then the sequencing is also called denovo sequencing and reads are aligned to each other to find the most appropriate position (called an assembly). The whole process is shown in Figure 1.2.

Sequence alignment is the process of comparing and detecting similarities between biological sequences. At first, the alignment subjects to heuristic algorithms. Dynamic programming giving us quadratic time complexity for two sequences is used by optimal aligning algorithms. Thus, to align a high number of reads it is not possible to align it in a reasonable time. An optimal alignment (usually Smith-Waterman or Needleman-Wunsch algorithm [43, 33]) is used for more sensitive alignment of smaller sections. According to a number of overlapping parts, a quality of a genome is measured and saved into the corresponding database.

1.9 Genetics variants

There is almost zero probability to have an identical genome even when comparing two organisms of the same species. Almost every genome is unique and contains some nucleotide changes, which are called variations. Many of these variations are nonsense or have no harmful effects on organisms because the gene expression has a very sophisticated repair mechanism. Most of them are created in the prenatal part of life, the rest of them are caused by mutation factors or by natural gene translation during the whole life. Two organisms could share the same variants (for example a blood group) and some variants could cause several kinds of diseases (Sickle cell disease).

Genetics variants are stored in specific format called variant calling format

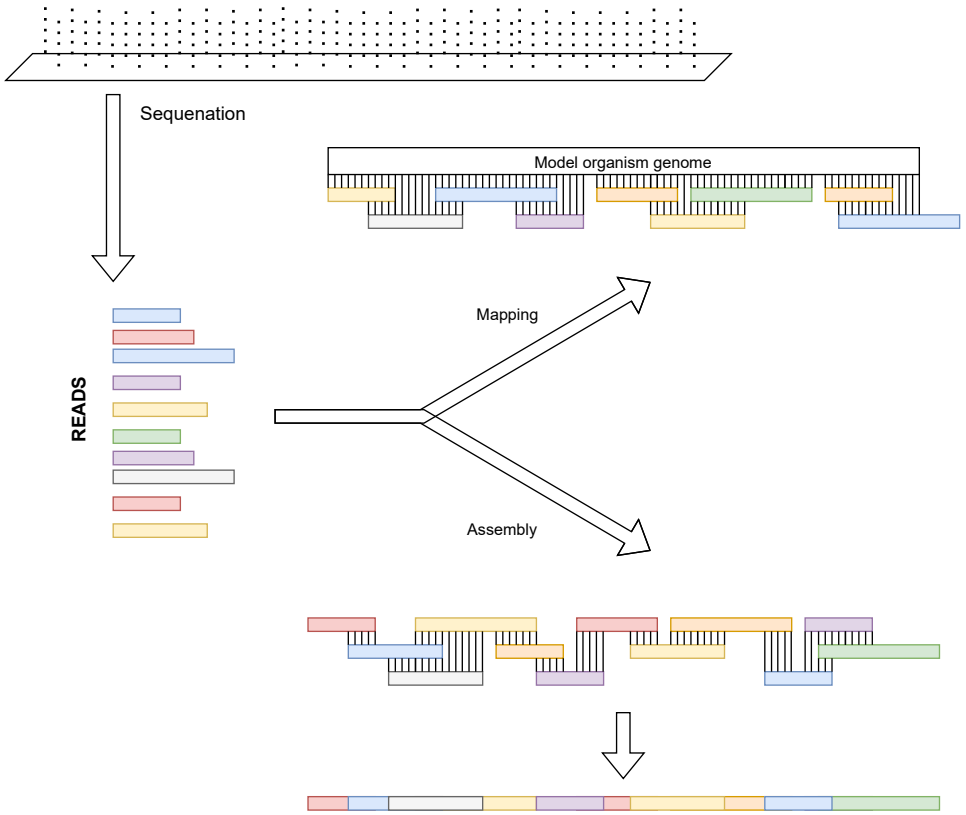


Figure 1.2: Short introduction to the basic methods accompanying sequence analysis from the raw short sequences in a sample to assembled sequence

(VCF) [35]. A pan-genomics picture could be formed from variant schemes. At the same time, the structure of a pan-genome has the potential to replace still used comparison with one "reference" genome, which has limitations of observation. Querying into a pangenomic structure is complex not only from the biological side but also from the stringology side and its structure is very complex. These days, the pan-genome structure is indexed and visualized using variant graphs, but there is a quite pressure from the scientific community to improve it.

More about VCF format and its hierarchy will be mentioned later in Section 2.4.

1,000 human project

The 1,000 Genomes Project includes a catalog of common human genetic variation, using openly consented samples from people who declared themselves to be healthy. The reference data resources generated by the project remain heavily used by the biomedical science community [10].

Variant graphs

A variation graph, defined in [18] is s graph with embedded paths
$G = (N, E, P)$, where N notes a set of nodes, E is a set of edges and P is a set of paths. Each node represents a sequence over alphabet A, C, G, R and is traversed in both directions. Edges represent connections between nodes and imply, that the sequence gained with the concatenation of node content can be much longer than the structure is. The P gives the variable number of executable paths, that specify the sequence encoded by a variant graph.

Chapter 2

Analysis and design

This section reviews the literature related to the BIO-FMI algorithm and its basic principles. We describe its structure, index construction, and index survey. We further develop the idea and try to apply it to a format containing elastic-degenerate strings. For every algorithm, the time and memory management analysis will be described.

2.1 Main idea

Since we are talking about a set of genomes or another genetic material, it is obvious that there is some redundant information rate. The evolution of every organism has common ancestors, and therefore their genetic equipment is similar to a certain extent.

In an effort to store and process every genetic material in its natural sequential form, we have a huge amount of redundant information. The main principle of the BIO-FMI and many other indexes is to store data and search in structures, which require only a fraction of the required information. One group of indexes is reference sequence-based algorithms and their principle is based on storing changes against one reference string. The reference string is agreed on consensus sequence in the case of biochemistry and genetics, but it does not correspond to any particular organism.

2.2 BIO-FM index design

The first idea was suggested by P. Prochazka and J. Holub in Compressing Similar Biological Sequences Using FM-Index [40]. Their implementation shows a very appropriate pattern searching time, even though the experiments were performed over an artificial dataset and the implementation was limited by only one type of allowed change — substitution.

Due to the relatively complex character of the algorithm, we define additional and required structures.

Collection of strings

r lexicographically ordered strings $T_0, T_1, \ldots T_r - 1, r > 1, T_i \in \Sigma^*$ are called a collection of strings. We denote T_0 as a reference string and $T_i, i \neq 0$ as i-th non-reference string.

Example 2.2.1 (Collection of strings and its alignment)**.**

T_0:AAAAAGGACAGGA		T_0:-AAAAAGGAC--AGGA
T_1:TAGGACACACATAGGA		T_1:TAGGACACACATAGGA
T_2:AGAGACACATAGGA	$\Rightarrow$	T_2:--AGAGACACATAGGA
T_3:AAAAGATAGG		T_3:--AAAAG---ATAGG-
T_4:AAAGAGAGGA		T_4:---AAAGAG---AGGA

Parameter *context_length* is defined as the constant value of stored information from the reference string. The parameter is also used to chop the input pattern into chunks of context length to search for patterns by parts, which will be explained later in Section 2.2.3.

Change in non-reference string

A change notes some difference between a reference and non-reference string. Technically, every non-reference string can be written as successive sequence of common (C) and difference (R) blocks. Therefore, $T_i = R_{i,0}C_{i,0}R_{i,1}C_{i,1} \ldots C_{i,k_i-1}R_{i,k_i}$ and R blocks can be called as changes.

From a genetic point of view, changes can be divided into many groups as referred to in Section 1.7. For our purpose all changes can be divided into the following three types:

1. **Substitution**: also called Single nucleotide polymorphism (SNP) is a change where one character is substituted for another. The offset against the reference string is zero.

2. **Insertion**: change, where in non-reference is inserted one or more characters. The length of the non-reference string is now bigger than the reference string and also the offset grows positively.

3. **Deletion**: change, where the substring is removed from the reference string in the non-reference string. The length of the non-reference string is now smaller than the reference string and also the offset grows negatively.

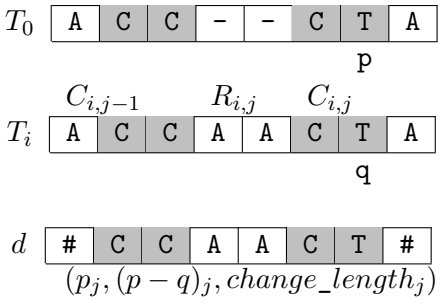
2.2.1 Change saving

As was mentioned in the main idea, we will store differences against the reference string. An assumption to store this kind of information is its corresponding input format. There are a lot of alignment algorithms, mostly based on dynamic programming, which can align biological sequences. After this adjustment, the sequences contain a character '-' as in Example 2.2.1. For our purpose, it means better parsing and detection of each change. The detailed discussion on input format will be included in Section 2.4. Example 2.2.2 describes storing process of each type of a change.

Example 2.2.2 (Change types)**.**

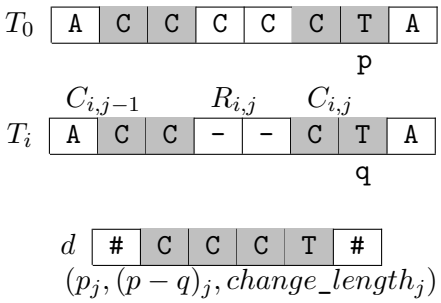
context_length = 3

a)



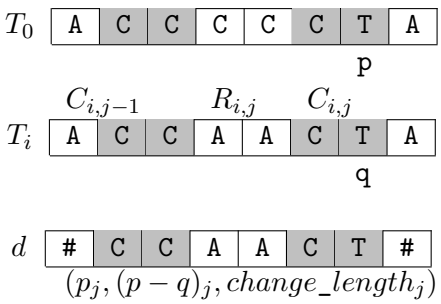
context_length = 3

b)



context_length = 3

c)



*Examples of types of changes: a) Insertion of "*AA*" substring, b) Deletion of "*CC*" substring, c) Substitution of "*CC*" substring to "*AA*".*

The changes are stored in the vector d as a concatenation of the changes separated by symbol #. Together with the changes, the left and right contexts of corresponding one less length of *context_length* are stored. Furthermore, there are additive structures that store the following values for every change:

1. **Base position**: A position in the reference string that corresponds to the end of the change.

2. **Offset**: A difference of ending positions in reference and non-reference strings.

3. **Change length**: The length of the change, including contexts. A substitution keeps lengths against the reference string, insertion is in positive numbers, and deletion in negatives.

Example 2.2.3 (Required structures for index build – summary)**.**
$context\_length = 3, number\_of\_changes = 10, number\_of\_sequences = 5$
T_0 : AAAAAGGACAGGA\$
d : #TAGGACACAC#ACATAG#AGAGACAC#ACATAG#AA#AAAGATAG#GG$#AA#AAAGA#GAGAG#
loc : 1000000000010000001000000001000000100100000000100010010000001000001
$iloc$:1000000000000000001000000000000000100000000000000100000000000001
$base\_position = \{8, 10, 8, 10, 3, 10, 13, 4, 7, 10\}$
$offset = \{-1, -3, 1, -1, 2, 2, 3, 3, 3, 3\}$
$change\_lengths = \{10, 6, 8, 6, 2, 8, 3, 2, 5, 5\}$
$start\_positions = \{0, 18, 34, 50, 65\}$

In our example, we get the required information and additive structures. The above mentioned example depicts another two structures of bit-vectors *loc* and *iloc*. These are used for quick rank and select queries.

2.2.2 Index build

The major part of the index building is divided into several steps. At first, transformations of strings T_0 and d are created follows by assembly of wavelet tree FM indexes. The structure follows algorithm complexity in Algorithm 3. Next, the index structure is sampled with *iloc* and *loc* vector, and rank and select structures are created. It is also necessary to build structures for rank and select constant time queries as in Algorithm 1 and Algorithm 2. A summary is marked in Algorithm 4.

2.2.3 Search in index

At first, the pattern is divided into k-mers of *context_length* size. For such long chunks, it is ensured, that the k-mer will be found in the reference in-

Algorithm 4: Building the indexes

1 function build $(T_0, d, iloc, loc)$;
Input : Two strings: reference T_0 and d (string of changes), two bit vectors: *iloc* and *loc*
Output: (I_0,I_d, $rank\_support$, $rank\_support$, $select\_support$)
2 $T_0^{BWT} \leftarrow createBWT(T_0)$;
3 $d^{BWT} \leftarrow createBWT(d)$;
4 $I_0 \leftarrow createWT\_FMindexT(T_0^{BWT})$;
5 $I_d \leftarrow createWT\_FMindexT(d^{BWT})$;
6 $riloc \leftarrow rank\_support(riloc)$;
7 $rloc \leftarrow rank\_support(rloc)$;
8 return $I_0, I_d, rloc, riloc$;

dex (in index I_0), or if the k-mer differs at least one character, then in the concatenation of changes d (in index I_d).

Then, the location using the *FM-Search* algorithm has been found in I_d and for each location, the correct pointer to the corresponding non-reference sequence and location has to be found. There can be easily found sequence and change numbers using rank queries into *loc* and *iloc* vectors. The corresponding position can be deduced by the base position in reference and an offset against the non-reference sequence and stored in a hash table.

Searching in the reference index follows a different procedure. We need to iterate all non-reference sequences and check the corresponding positions for non-stored parts of the sequence. If there is a non-changed substring on the position, it has to be stored in the hash table too. It means, we need to iterate all sequences for each location found in index I_0.

This process is repeated for each next chunk with the difference, that every non-first chunk has to be validated according to stored locations in the hash table. Algorithm 5 shows the whole process of pattern search.

2.2.4 Special cases

During the whole process, there are different situations. At first, what happens if two changes are too close to each other (their distance length is smaller than the stored context) – then the two changes are merged into one bigger and sometimes it might happen, that we cannot recognize the change type anymore.

If the changes are too close to the end or the beginning of the sequence, then there have to be additive conditions. In this case, there is more than one way to implement it depending on base positions p and q. In this document, we work with appending of symbol `$` at the end of each sequence to detect correct ends.

Algorithm 5: Search in indexes

```
1  function search (P, m, I_0, I_d, r, context_length);
   Input  : Pattern P of size m, indexes I_0 and I_d, additive structures
            base_position, offset, change_lengths
   Output: (I_0, I_d, rank_support, rank_support, select_support)
2  chunks ← parse P_{1,m} into chunk of length context_length;
3  for i ← 1 to |chunks| do
4      occ ← FMSearch(I_d, chunks_i);
5      for j ← 1 to |occ| do
6          sequenceNumber = rank_iloc(occ_j);
           changeNumber = rank_loc(occ_j);
           position_j = basepositon_j + offset_j + differenceFromHash_j;
           if isValid(position_j) then
7              store position_j to a hash table;
8          end
9      end
10     occ ← FMSearch(I_0, chunks_i)
11     for j ← 1 to |occ| do
12         if isValid(occ_j) then
13             store occ_j to a hash table;
14         end
15         for k ← 1 to r − 1 do
16             position_j = find the corresponding position and upcoming
                changes;
17             if isValid(position_j) then
18                 store position_j to a hash table;
19             end
20         end
21     end
22 end
23 Report all positions with complete P1, m in a hash table;
```

2.2.5 Time and memory management

Let us recall that n is the length of input text. Then the reading and parsing are done in linear time. The build time is given by the time and memory complexity of the FM-index wavelet tree, which can be built-in $\mathcal{O}(n)$ time.

To find a pattern (a chunk in our case) of length m_j in FM-index takes $\mathcal{O}(m_j \log |\Sigma|)$. In the worst case, the index finds occurrences in every position in the text, and iterating them takes extra $|occ| * number\_of\_sequences$ time, where the $|occ|$ is a number of occurrences found by the FM-index. The validation is in constant time and the whole process of searching is done

exactly $\lceil \frac{m}{context\_length} \rceil$ times.

Finally, we get $\mathcal{O}(\lceil \frac{m}{context\_length} \rceil * |occ| * number\_of\_sequences)$ linear time complexity at all.

Wavelet tree FM-index takes $\mathcal{O}(\log n)$ bits of storage. In the worst case, the concatenation of changes will be the same size as the original text plus constant for surrounding context and hash symbols. This case is not probable, because we discuss highly repetitive texts. The size of additive structures (*base_position*, *offset*, and *change_lengths*) depends on similarity of sequences.

2.3 BIO-FM Index design over EDS

In view of the EDS format, the substantial part is its parsing into the required format. It is appropriate to remind indexing way in EDS as mentioned in Definition 1.1.8. Let's assume, that the sequences in each block are ordered and the reference string is composed of the shortest sequences in each elastic degenerate symbol. Then the position corresponds to the position in EDS with minimal length, and when the change appears in some block against the reference sequence, the position is shifted according to the offset between the change and reference string.

2.3.1 Change saving

When two elastic degenerated symbols are too close, then the new block of size $|\xi|_i * |\xi|_{i+1}$ is created using Cartesian multiplication. During this process, it may happen that some sequences have a common prefix/suffix with the reference sequence. An offset of change is the difference between lengths of sequence used in the reference string and sequence in an actual change (offset is always a positive number). In *start_positions* pointers are stored to the d string to the next segment hash symbol.

An example of EDS parsing into BIO-FMI format is described in Example 2.3.1.

Example 2.3.1 (Parsing of EDS)**.** $T = \texttt{AB} \begin{bmatrix} \varepsilon \\ \texttt{C} \\ \texttt{CC} \end{bmatrix} \texttt{BAC} \begin{bmatrix} \texttt{AA} \\ \texttt{RA} \end{bmatrix} \texttt{RA}$

$context\_length = 5$

$$T \approx \texttt{AB} \begin{bmatrix} \texttt{BACRA} \\ \texttt{CBACAA} \\ \texttt{CBACRA} \\ \texttt{CCBACAA} \\ \texttt{CCBACRA} \end{bmatrix} \texttt{RA} \qquad \Rightarrow \qquad T_0 = \texttt{ABBACAARA}$$

$d = \texttt{\#ABCBACA\#ABCBACRARA\$\#ABCCBACA\#CCBACRARA\$\#}$

$base\_position = \{5, 9, 5, 9\}$
$offset = \{1, 1, 2, 2\}$

2.3.2 Index construction

According to recreating the input format into BIO-FMI format, the index construction and construction of additional structures are same as in Algorithm 4.

2.3.3 Searching in the index

At first, let's take a look at a result conception of locate query. One of the ways used in this document is to set location with increasing sequences of the absolute number of changes. It means, that we do not take care about the number of blocks, but only the overall order of sequence in EDS. See Table 2.1.

$position_0$:	[	$E_{x,y}$,	...	$E_{x+k,y+l}$]
⋮				
$position_p$:	[	$E_{x,y}$,	...	$E_{x+k,y+l}$]

Table 2.1: EDS output format visualization

Searching in indexes is similar to the algorithm in the previous design in Section 2.2.3. At first, the pattern is divided into the chunks of the corresponding length. These chunks are searched for in I_0 and I_d in any order and subsequently iterated. To find the corresponding block and change the number, rank queries over *loc* and *iloc* bit-vectors are used.

Every next chunk is validated with existing occurrences. Therefore for every location of chunk and for every saved preceding occurrence, it is validated for both reference and non-reference positions. This validation is essential, especially for verifying non-conflicting sequences of used sequences in blocks. The whole process is described in Algorithm 6.

2.3.4 Time and memory analysis

The time complexity of index build is the same as in the previous analysis. In contrast with it, the size of text can increase depending on the segment organization and context length. The time for finding occurrences in the FM-index wavelet tree takes $\mathcal{O}(m_j \log |\Sigma|)$, where m_j is the length of the chunk. The maximum number of occurrences n in the worst case, but we do not assume it, so note $|occ|$ as a number of occurrences founded by the FM-index Validation of one occurrence takes in the worst case $\mathcal{O}(|occ|_{j-1})$ time complexity, where $|occ|_{j-1}$ is a number of valid occurrences from the previous chunk. The whole process is repeated $\lceil \frac{m}{context\_length} \rceil$ times. Finally we get estimation of asymptotic complexity $\lceil \frac{m}{context\_length} \rceil * (\mathcal{O}(m_j \log |\Sigma|) + \mathcal{O}(|occ|) *$

Algorithm 6: Search in indexes over EDS

```
1  function search (P, m, I_0, I_d, r, context_length);
   Input  : Pattern P of size m Indexes I_0 and I_d, additive structures
            base_position, offset, change_lengths
   Output: (I_0,I_d, rank_support, rank_support, select_support)
2  chunks ← parse P_{1,m} into chunk of length context_length;
3  for i ← 1 to |chunks| do
4      occ_d ← FMSearch(I_d, chunks_i);
5      occ_0 ← FMSearch(I_0, chunks_i)
6      for j ← 1 to |occ_d| + |occ_0| do
7          changeNumber = rank_loc(occ_j);
8          segmentNumber = rank_iloc(occ_j);
9          for k ← 1 to |preceding_occ| do
10             if isValid(position_j) then
11                 Append current changeNumber to a valid start
                    position;
12             end
13         end
14     end
15 end
16 Report all positions with complete P1, m in table;
```

$\mathcal{O}(|occ|)) \approx \mathcal{O}(|occ|^2)$. This time complexity is only a theoretical assessment and the real-time complexity decides the experimental results. Furthermore, the set of $|occ|_{j-1}$ is with every iteration smaller until the positions for a full pattern stays alone.

The space complexity is given by the complexity of the wavelet tree and localization overhead as in the previous design. On the other hand, the content of *offset* array gives us information in change lengths and that means there is no need to store *change_lengths* anymore, and memory complexity is smaller.

2.4 Input format discussion

A sequence alignment based on biological similarity is really time-consuming problem. These algorithms are mostly based on dynamic programming and only pairwise alignment gives us an optimal result in quadratic time. For our purpose, there is no need to have an optimal alignment, but still, we need some information about the sequence similarity. This information is provided in a few bioinformatics formats.

The most common format to save some biological material is probably FASTA format. [31]. FASTA has no additive information about sequence except for the name of the sequence and information stored in the name.

This format is very often able to be accepted by many bioinformatics tools as input. Similar to FASTA is FASTQ, where always the first of the four is a string about sequential quality and eventual comments [11]. In both of these files, only the sequences in their raw form are available.

Short reads, aligned and mapped against reference sequence are stored in SAM file with its binary form — BAM. SAM and BAM files contain quite a big amount of information. However, this format does not contain solid sequences. On the contrary, there are amounts of overlapping sequences.

The best potential input format is the variant call format (VCF) presented by Petr Danecek et al. [34] in 2011. These file stores variants as an enumeration of changes against a reference genome, usually saved in FASTA file. The number of variant types is described in the header of the file together with reference links and basic information about organisms. The body of the file consists of basic eight columns and continues with columns of each sample organism, that is included in the file. This sample organism has binary pair information about change occurrence in each chromosome. See the example in Figure 2.1.

VCF is a good format to store intergenomic information, but its indexation is challenging. These days, the variant graph is used for indexation and visualization, but in the future, there is a great potential in this field to improve the time complexity and visualization of pangenomatic research. Moreover, the whole concept based on the reference sequence could be misleading in research over it, so another approach of storing and working with a set of genomes is welcome.

In our algorithm, the alignment information is needed, but moreover, the surrounding context has to be stored. It would lead to heavy parsing between opened reference files, which size could be enormous, and VCF. Finally, in the VCF format, there are variant types of tandem elements that are hard to explore and their exact sequence is not available in this format. It could be probably tracked in some databases, but this approach leads to a more complex issue. VCF and SAM can be processed with a set of programs called Samtools [6].

To summarize, the simplest way is to load a suitably large FASTA file and align it using a good heuristic tool for multi-sequential alignment. The use of such an approach is small because it is not used in practice. The most appropriate approach is to hardly parse the VCF format, separate all required information and run it. There was not enough time and resources in this work to deal with this problem in more detail.

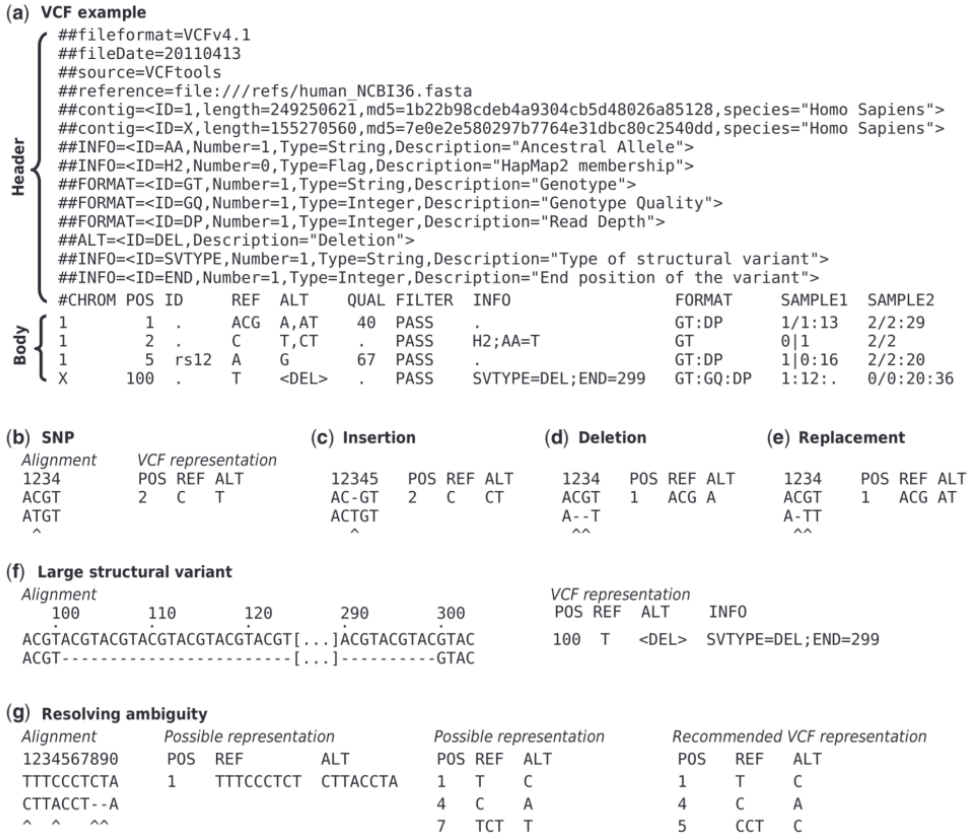


Figure 2.1: **(a)** Example of valid VCF. The header lines `##fileformat` and `#CHROM` are mandatory, the rest is optional but strongly recommended. Each line of the body describes variants present in the sample population at one genomic position or region. **(b-f)** Alignments and VCF representations of different sequence variants: SNP, insertion, deletion, replacement and large deletion. The REF columns show the reference bases replaced by the haplotype in the ALT column. The coordinate refers to the first reference base. **(g)** Users are advised to use the simplest representation possible and the lowest coordinate in cases where the positions are ambiguous [34].

Chapter 3

Implementation

This chapter will describe the structure of the application, the libraries used, and the input format. Next, the implementation of LZ-RLBWT and r-index, essential algorithms for our experiments, will be briefly described, and finally, we describe help scripts used for data generations and result processing.

3.1 Overall structure

The structure of our tool is quite simple. It consists of two small applications, the structure and communication with the filesystem are depicted in Figure 3.1. Application organization is inspired by the implementation of LZ-RLBWT and r-index, which have simple usage and organized outputs.

3.2 Technologies and libraries

The applications are written in the C++17 program language. As additional libraries, the boost library [4] and the Succinct Data Structure Library Lite [20, 19] have been used. Both of these libraries are freely available. The choice of SDSL library was clear, because of the effective implementation of many structures including wavelet trees, FM-index, and rank and select support structures over a bit-vector. The classes from SDSL library have been used for my implementation are shown in Table 3.1.

The boost library was used for the algorithmic work with strings and input parsing. The helpful scripts are written in Python 3.8 language.

3.2.1 LZ-RLBWT

LZ-RLBWT is an algorithm designed in [3] and implemented by Nicola Prezza et. al. It is free to download and use on GitHub repository [38]. The experiments have their own website with the required information and more on Pizza&Chilli website [36]. LZ-RLBWT is an algorithm-based combination of

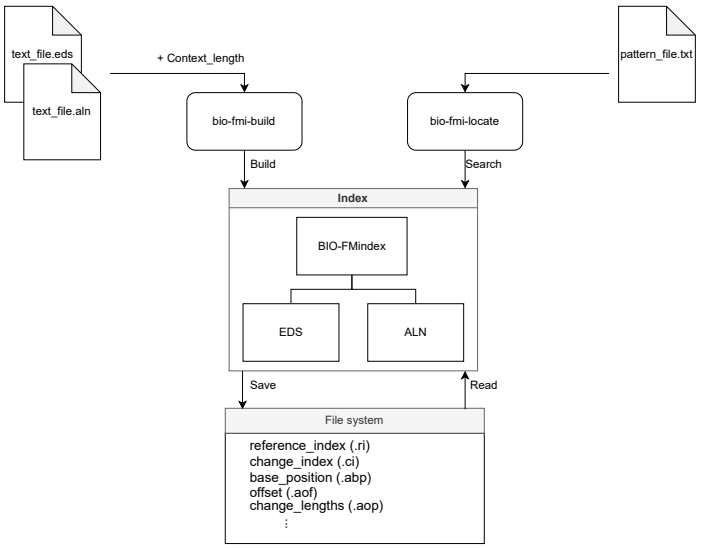


Figure 3.1: The structure of applications and their communication with the filesystem. On the left the input files for index construction application – *bio-fmi-build*, on the right side pattern file with index construction path as input for *bio-fmi-locate* application. The index is stored in the filesystem with a number of files stored $I_0, I_d, base\_positions, offset, change\_lengths, loc, iloc$ and configuration settings

WT FM-Index	`csa_wt < t_wt, t_dens, t_inv_dens, t_sa_sample_strat, t_isa_sample_strat, t_alphabet_strat >`
Select	`select_support_mcl < t_b, t_pat_len >`
Rank	`rank_support_v < b, pattern_len >`

Table 3.1: SDSL classes; A class for the Compressed Suffix Array (*CSA*) based on a Wavelet Tree (*WT*) of the *BWT* of the original text, that corresponds to FM-index structure; Select structure supports constant time queries with additional $\leq 2 * n$ bits. Rank structure supports constant time queries with additional 64 bits.

the Lempel-Ziv (LZ) algorithm over run-length BWT transformation of the original string. The structure takes approximately $\mathcal{O}(z + r)$ memory, where z is the number of factors in the LZ parser and r is the number of runs in BWT. The index reports all the occurrences of a pattern of length m in $\mathcal{O}(m(\log\log n + \log z) + pocc \log^{\epsilon} z + socc \log\log n)$ time, where n is the length of the string and *pocc* and *socc* are the number of primary and secondary occurrences. See the definition of primary and secondary occurrences in the original text of the Lempel-Ziv algorithm [25].

3.2.2 r-index

The algorithm of r-index was described in 2017 and improved in 2020 [17, 16]. An implementation by Nicola Prezza is free to download and use on its gitHub repository [39]. The r-index is the first full-text index of size $\mathcal{O}(r)$, where r is the number of BWT runs of the input text of size n. This algorithm reduces the time of FM-index locates query from $\Omega(n/r)$ to $\mathcal{O}(\log(n/r))$ and takes $r*(\log|\Sigma|+(1+\epsilon)\log(n/r)+2\log n)$ bits, where epsilon is a non-zero constant value. For experiments details over the r-index I refer to Pizza&Chilli testbed website [36]. The implementation accepts raw *.txt* files with every sequence on a separate line.

3.3 Application setting, input format, and workflows

Application *bio-fmi-build* has one optional, but highly recommended parameter — *context_length*. This value is set by default to value six (justified in Chapter 4). The text input format can be used by files with *.aln* and *.eds* extensions. These formats are not used in common practice — this topic was already discussed in Section 2.4. An *.aln* and *.eds* formats are depicted in Figure 3.2.

a) ALN input format

```
-AAAAAGGAC--AGGA
TAGGACACACATAGGA
--AGAGACACATAGGA
---AAAAG--ATAGG-
----AAAGAG--AGGA
```

b) EDS input format

```
AG{,CC,GGG}CAAAAA{AA,TA}TA{,CC,TA}
```

Figure 3.2: The example of text file formats: **a)** a set of aligned sequences, required the same lengths and the first line is taken as a reference string. **b)** EDS format, where each elastic degenerate symbol is separated by braces. The reference string is concatenation of seeds and first sequence from each block.

The implementation has an adjustable output path and silent mode, which does not print parsed text files. The whole usage of the application is intuitive and very simple (see Figure 3.3). For more detailed usage of the applications, see Chapter B.

3.4 Format of index storing

For this time, every index and additive structure have been saved in a solo file. The implementation creates a folder with the corresponding name and

a) Build application usage

```
$ ./bio-fmi-build --help
Usage: ./bio-fmi-build [options] <input_file_name>

Allowed options:
  --help                        produce help message
  --help-verbose                display verbose help message
  -v [ --version ]              display version info
  -s [ --silent ]               silent mode
  -r [ --repetition ] arg       number of repetition - for experiment needs
  -l [ --context_length ] arg   length of chunk and stored context, default 5
  -o [ --basefolder ] arg       use <basefolder> as prefix for all index files.
                                Default: build folder is the specified
                                input_file_name
  -i [ --input-file ] arg       input file
```

b) Locate application usage

```
$ ./bio-fmi-locate --help
Usage: ./bio-fmi-locate [options] <index_basename> <pattern_file>

Parameters:
  -h [ --help ]              display help message
  --help-verbose             display verbose help message
  -v [ --version ]           display version info
  -s [ --silent ]            silent mode
  -p [ --pattern ]           print occurences of every pattern
  -i [ --index-path ] arg    input text file path (positional arg 1)
  -I [ --pattern-file ] arg  input pattern file path (positional arg 2)
```

Figure 3.3: Usage of **a)** build and **b)** locate applications

subsequently saves both indexes I_0, I_d and all additive structures from Example 2.2.3. This way of implementation uses the function of SDSL library (*store_to_file()*), which can save any structure using XML language and vice versa to load the structure using the function *load_from_file()*. In the future, the complex structure of additive arrays could be stored more efficiently in some kind of compressed form.

The rank and select support structures cannot be saved with the functions mentioned above. For this time, the structures are transformed after loading of the whole structure including bit-vectors. There was no problem with the transformation of these structures because their construction is relatively fast.

3.5 Data generators

In this section, the characterization and setting of data generators are described. The scripts were used for generating pseudo DNA dataset over alphabet $\Sigma = \{A, C, G, T, N\}$ and corresponding pattern files. Let us recall that even if N is used in our alphabet to mark any of the nucleotides, the implementation and algorithm support only the exact pattern matching yet.

The first script, *create_pseudoDNA*, is used to generate a random multi-sequence alignment without a link to a biological characterization. The output of the script is in the format *.aln* described above. It has three different parameters at the entrance — the number of sequences, the total size of the file, and the probability of change against the reference sequence. The total size is finally larger than its input parameter because the size is given by all characters in the file, including the symbol '-' for the gap, while the input parameter gives the amount of raw information in DNA. By removing this *gap symbol* we get an input file for LZ-RLBWT and r-index applications.

Within the Pizza&Chilli website, several helpful scripts for working with data are available. According to this, the *genpatterns* script was used to generate random patterns from the file. The number and length of the pattern are required parameters. This script prevents the search for patterns that are not available in the text file.

For generating random text files in format *.eds*, the script taken over SOPanG [8] was annotated and used. For our experiments, the *create_eds* script accepts the required total size of the file and the probability of change, which gives a suitable number of blocks in EDS. The total size is again quite larger according to the symbols that mark the start and end of the elastic degenerate symbols.

The last script *genEdsPatterns* is used for the pattern extraction from EDS file format. The script requires several patterns and lengths together with input and output file paths. The script is used to prevent the search for non-existing random patterns in the text file.

The implementation of BIO-FMI and additional data generators are freely available in my GitHub repository [13].

Chapter 4

Experiments

The experiments are divided into four main parts depending on the dataset and the aim of the measures. Note that the implementation is designed for a simultaneous run with one thread.

The number of results is quite high, which means that in this chapter we describe only a fraction of them. See Appendix A for all measurements and visualisations.

4.1 Measuring environment specifications

All experiments were carried out on the server covered by the Department of Computer Science at CTU Prague with the following hardware statements.

OS	GNU/Linux Kernel 5.15.41
CPU model	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
RAM memory	32GiB
Swap memory	29GiB

Table 4.1: Environment specifications

4.2 Time measurement

In order to measure accurately and omit the influence of other processes, the measurement technique has to be adapted. The total CPU time contains user time, which gives us the time spent in our code and system time, which indicates the time eaten in kernel space with operations like the work with files, kernel calls, etc. The total time is given by the sum of system and user times and for this purpose function *getrusage* from the system library of C with variables *ru_stime* and *ru_utime* was used. The best evaluation of the results can be obtained with the repetition of experiments to suppress noise from

other processes. We use a higher number of patterns for one run to get the most valuable average results independently of the pattern characterization and the number of occurrences, and we observe the average search time per occurrence and per pattern.

4.3 Memory measurement

It is almost impossible to obtain an exact measurement of the memory usage of a program and, simultaneously, there is usually no need to know the complete usage of memory. The major interest of our experiment is in the higher value of consumed RAM memory during the process running (Peak RAM memory usage). The value can be measured using the function *getrusage* again, and we are asking for the maximum size of our resident set, hidden under a variable called *ru_maxrss*.

Unfortunately, memory measure during the pattern matching part is unstable. The searching process lasts too short of time to get some adequate measurements, so this document does not contain memory comparisons during the pattern matching.

4.4 Datasets overview

Experiments were running over pseudo-real data most time due to already discussed problems in Section 2.4. But the result over real data is very important for us, we show two real sets of DNA sequences, which are small enough to gain their multi-sequence alignment. The pseudo-real dataset construction will be described in the following section. Table 4.2 shows all datasets and their characteristics, which all experiments were running over.

Experiment	Pattern length	PseudoDNA – .aln		
Dataset name	PL	ALNVLength	ALNVNumber	ALNVDifference
FIle size [MB]	1	$2^i, i \in [0, 8]$	4	4
Number of Sequences	100	100	$2^i, i \in [1, 9]$	100
Change probability [%]	10	10	10	1,10,25,50

Experiment	Thyroid peroxidase	Drosophila tripunctata	PseudoDNA – .eds	
Dataset name	TPO	DT	EDSVSize	EDSVNumber
FIle size [MB]	0.01402	0.09192	$2^i, i \in [0, 3]$	4
Number of Sequences	6	63	-	-
Change probability [%]	-	-	10	$1, 5, 10$

Table 4.2: Datasets overview

4.4.1 Pattern and stored context optimal length

Our algorithm is based on the iteration of all locations followed by concatenation with preceding chunks, for the speed, it is necessary to set the optimal length of chunks or the whole pattern. The frequency of occurrence of shorter patterns is bigger than for longer ones. It means, for too short patterns, their frequency will be enormous and it increases linearly against the text size and the algorithm consumes a lot of time.

From the perspective of biological science and its view of the optimal setting of *context_length*, for every amino acid, there are three nucleotides to code them and due to this, there is no need to search patterns shorter than 3. Moreover, the coding sequences begin with promoters and end with stop codons. There are at least nine characters with some information between them. Furthermore, there is a utilization without biological background. For example, searching for words that are used as precursors for other algorithms such as BLAST [1], or words for natural language processing (NLP) of DNA [28].

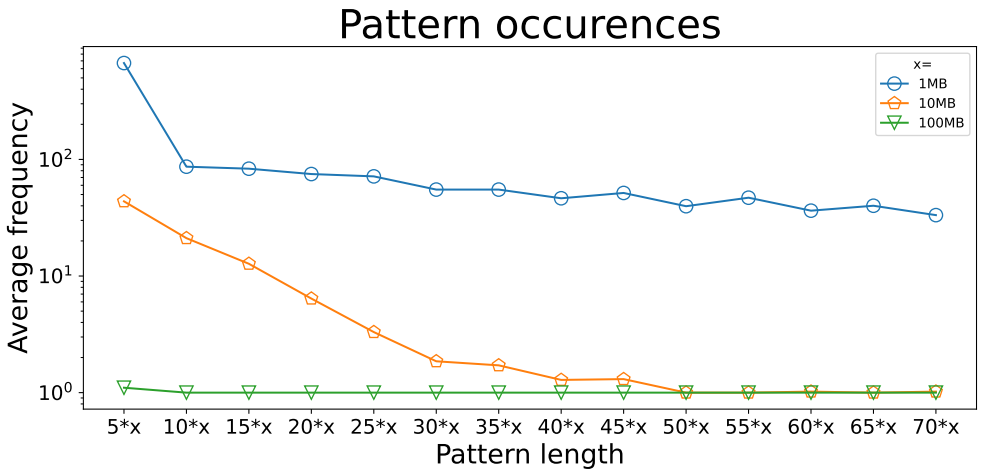


Figure 4.1: Average number of occurrences (logarithmic scale) per pattern of specific length in relation with input file size

For this purpose, dataset PL was created. It contains three input text sizes 1MB, 10MB, and 100MB. These data are scanned for pattern lengths $x \times text\_size$ for x in 5–75 with step 5.

Results are shown in Figure 4.1, where on the vertical axis are average frequencies with a logarithmic scale, and on the horizontal axis are plotted lengths of searched patterns in their relations to file size.

4.4.2 Experiments over pseudo-DNA datasets

The datasets were created using generator script *create_ALN* described in Chapter 3. The main parameter that could affect the algorithm's behavior

is the length of sequences, the number of sequences, and their similarity. We must concern that, the three main datasets were created in the relation to a variant parameter — ALNVLength, ALNVNumber, and ALNVDifference. These datasets were transformed to *.txt* files and used as input for other comparative algorithms.

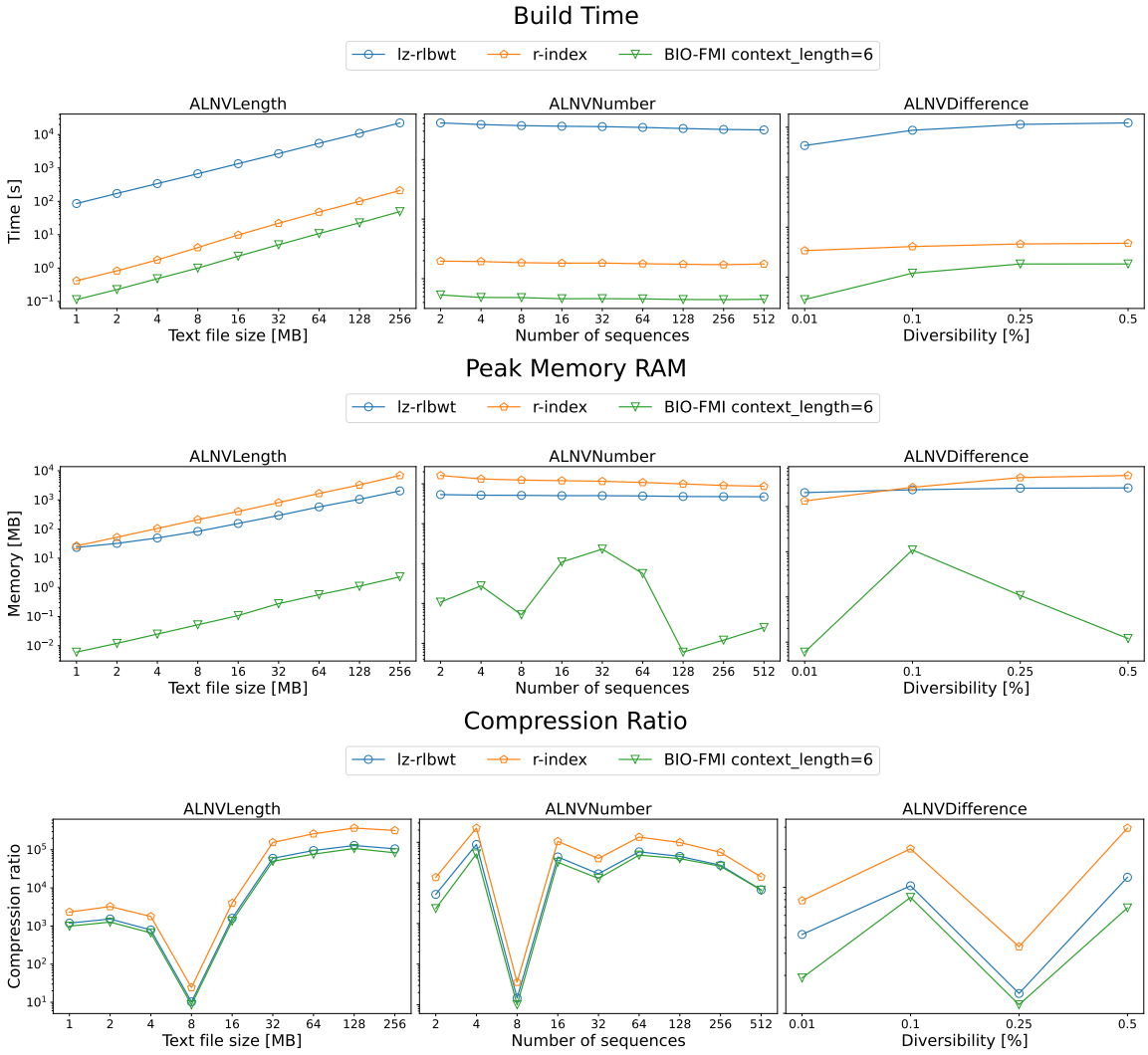


Figure 4.2: ALN datasets results review — Build time, peak RAM memory usage and compression ratio.

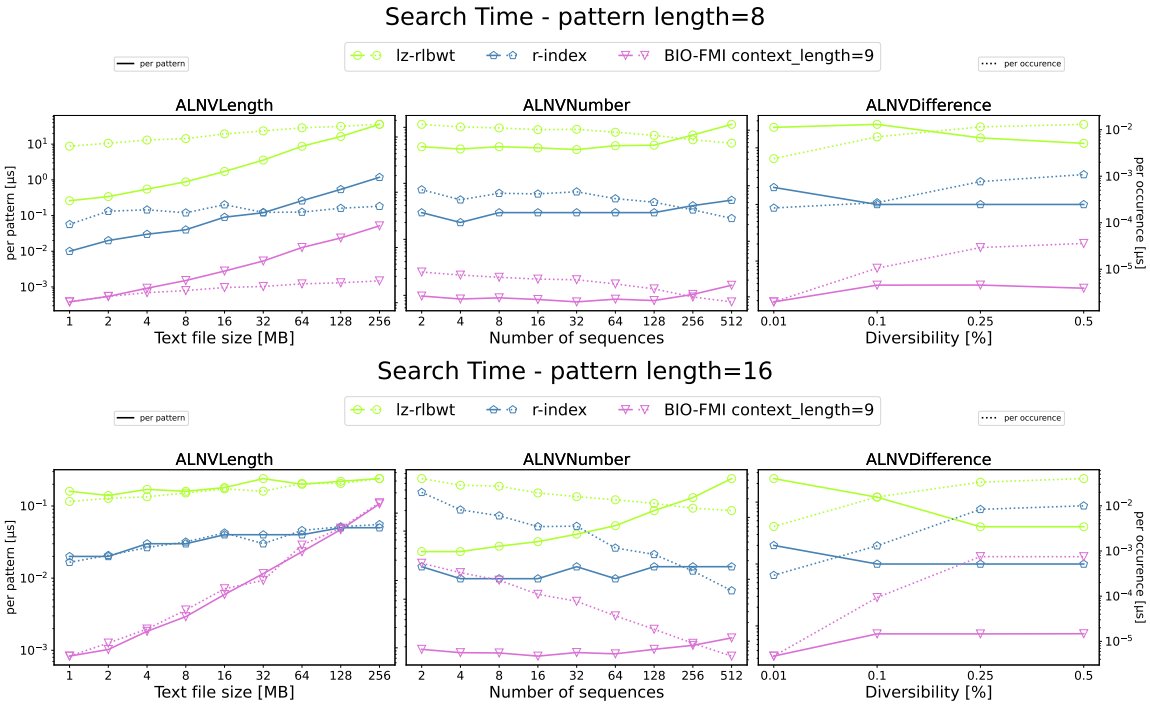


Figure 4.3: ALN datasets results review — Search time per pattern on left axis and per occurrence on right axis for length 8 and 16.

If you look at Figure 4.2 and Figure 4.3, you will notice that there is a comparison of three algorithms for every pseudo-DNA dataset o the horizontal axis, and the main observation parameters are the build time, peak memory RAM, and the compression ratio on the vertical axis. Search results have the search time per pattern (left vertical axis) and per occurrence (right vertical axis), which are the most valuable measurements. Figure 4.4 describes the influence of context length on build and search time with the same graph organization as was described for build and search graphs.

4.4.3 Experiments over real DNA datasets

The index was tested on two smaller datasets of real DNA sequences. At first, the six human variants of thyroid peroxidase isoforms were downloaded from NCBI GenBank [12]. These transcripts variants in mRNA form contain from three to four thousand base pairs. Let's note this dataset as TPO. The second dataset was downloaded from BOLDSYSTEM [41] and contains sixty-three sequences of Drosophila Tripunctata. There is a combination of COI-5P and COI-3P. Mark this dataset with the DT shortcut.

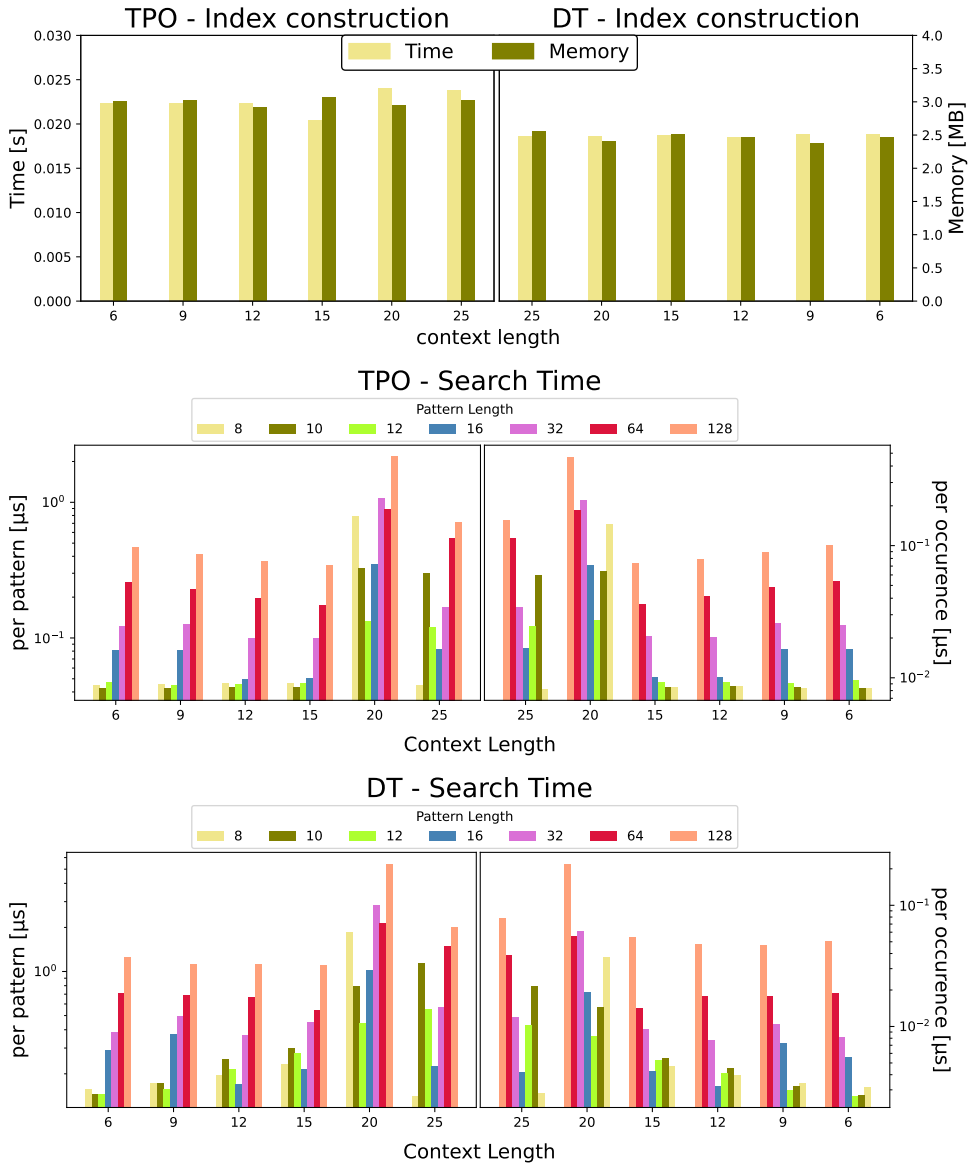


Figure 4.5: Experiment results over real datasets TPO and DT

The results over real DNA sequences in Figure 4.5 give us a good view into time and memory complexity, which is closer to bioinformatics practice.

4.4.4 Experiments with EDS over pseudo—DNA datasets

The datasets were created using generator script *create_EDS* described in Chapter 3. The format is able to store more information about several sequences in a smaller format, so there is no need to try the too big size files. We created text files in the dataset EDSVSize with sizes from ten thousand kilobytes to eight megabytes to test the behavior of the increasing size of the input file. The number of elastic degenerate symbols was deduced by the size of the input file to ten percent of changes.

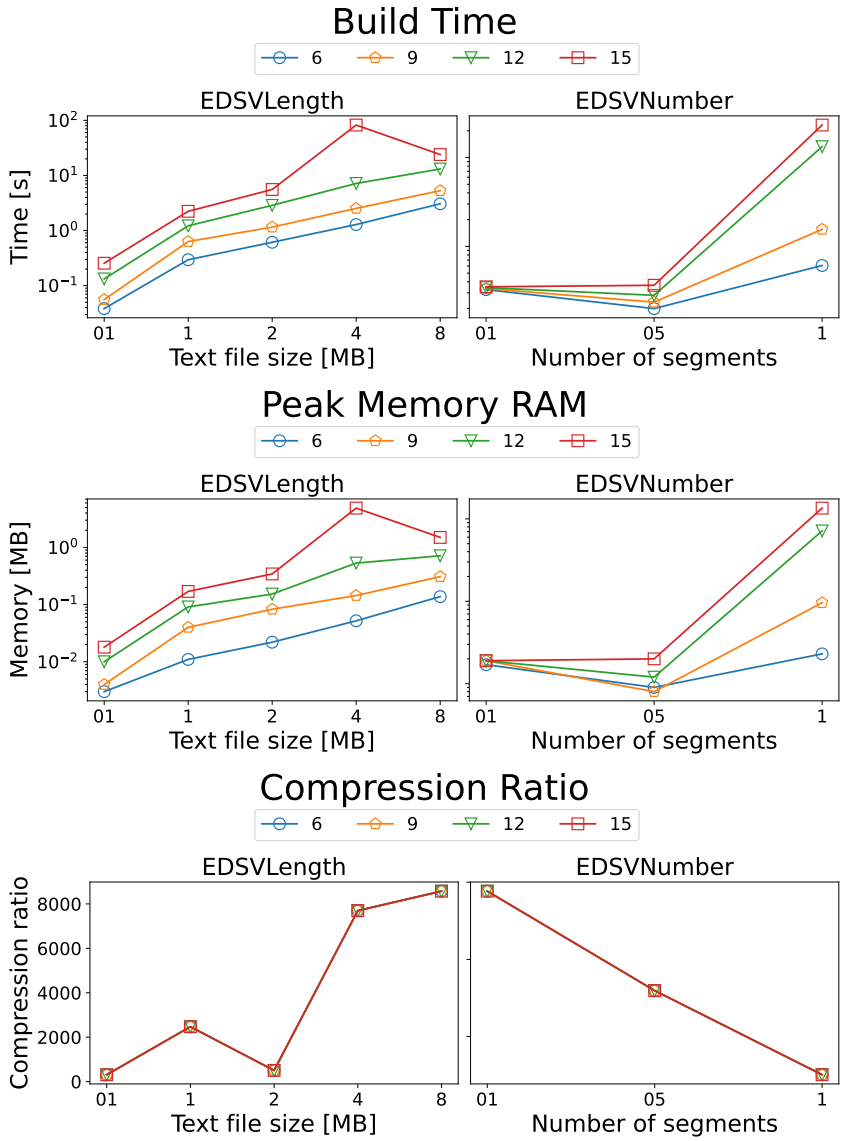


Figure 4.6: Construction results over EDS - Time, memory and compression ratio

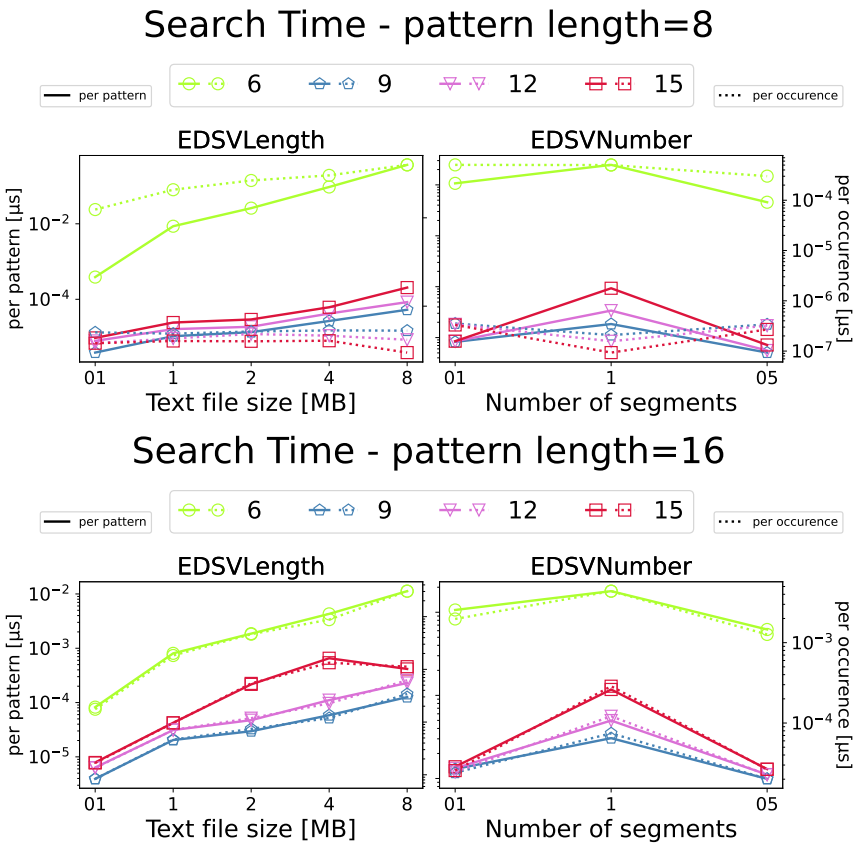


Figure 4.7: Search results for EDS datasets — time per pattern and per occurrence

In the second dataset EDSVNumber tests algorithm behaves according to the number of elastic degenerate symbols. It is necessary to note that a larger amount of ED symbols leads to a smaller space between them and it leads to their coupling and input text size increase.

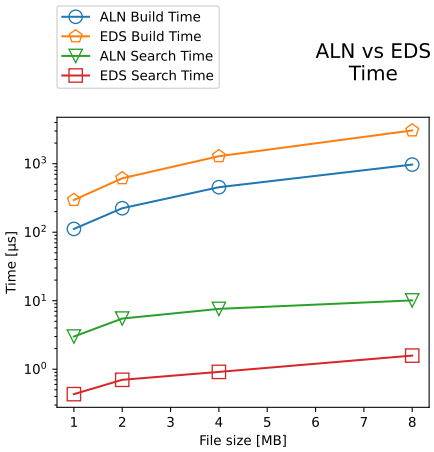
We can assume that higher context length settings and a larger amount of degenerate symbols can rapidly retard read and build time. This is the reason why the context length was limited to a value of 15.

Figure 4.6 shows the build time of described datasets and they are followed by a search comparison over the index in Figure 4.7.

4.5 ALN and EDS comparison

Based on the previous experiments, one of the most interesting deductions can be presented. We assume, that the behavior of the algorithm over *.eds* format will be quite worse either in construction and pattern matching. As described in the following graphs and tables, we compare them in input file size terms and also their reaction to pattern length and size of stored context length.

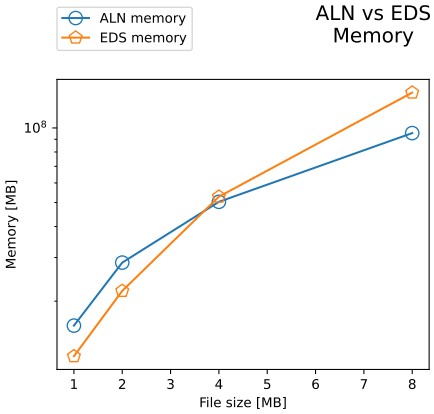
All measurements were taken with *context_length* 6 and over pattern of length 8.



	Build Time		Search Time	
	ALN	EDS	ALN	EDS
1	111.603	295.213	2.99896	0.43261
2	223.844	612.741	5.48827	0.70021
4	453.877	1286.010	7.60087	0.91363
8	971.139	3057.440	10.10840	1.57992

Figure 4.8: Comparison ALN and EDS in terms of construction time with logarithmic scale

Table 4.3: Comparison ALN and EDS in terms of construction time, original measurements



	ALN memory	EDS memory
1	15937000.0	11988000.0
2	28619000.0	22000000.0
4	50341000.0	52804000.0
8	95326000.0	138912000.0

Figure 4.9: Comparison ALN and EDS in terms of peak RAM memory with logarithmic scale

Table 4.4: Comparison ALN and EDS in terms of peak RAM memory, original measurements

Figures 4.8 and 4.9 together with Tables 4.3 and 4.4, respectively, describe the dependency on the file size. Vertical axes are in logarithmic scale and the tables contain the original measurements. At the first sight, the EDS gives very good result against the original implementation over *.aln* format.

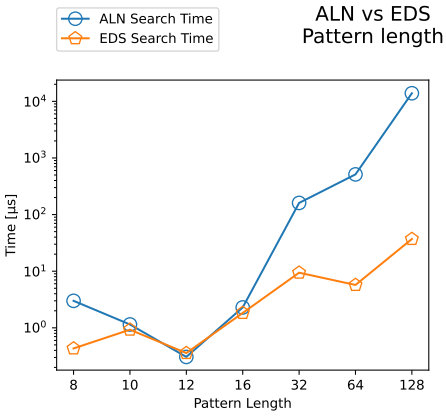


Figure 4.10: Comparison ALN and EDS in terms of pattern length with logarithmic scale

	ALN Search Time	EDS Search Time
8	2.99896	0.43261
10	1.14691	0.92726
12	0.30633	0.35864
16	2.29278	1.83122
32	160.64700	9.41601
64	512.67900	5.71631
128	13895.40000	37.04910

Table 4.5: Comparison ALN and EDS in terms of pattern length, original measurements

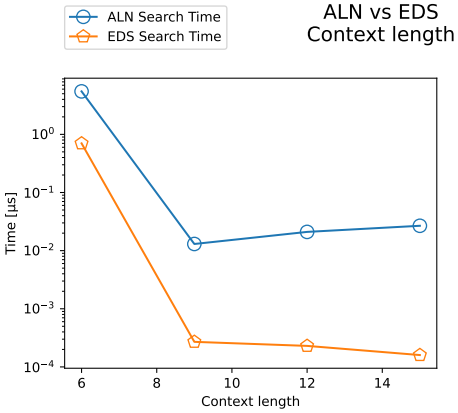


Figure 4.11: Comparison ALN and EDS in terms of context length with logarithmic scale

	ALN Search Time	EDS Search Time
6	5.48827	0.70021
9	0.01302	0.00027
12	0.02101	0.00023
15	0.02677	0.00016

Table 4.6: Comparison ALN and EDS in terms of context length, original measurements

Figure 4.10 and Table 4.5 show comparison in terms of dependency of search time on pattern length. The second pair, Figure 4.11 and 4.6 represent dependency on length of stored context. The search time is in both graphs in logarithmic scale and the tables contain original measurements.

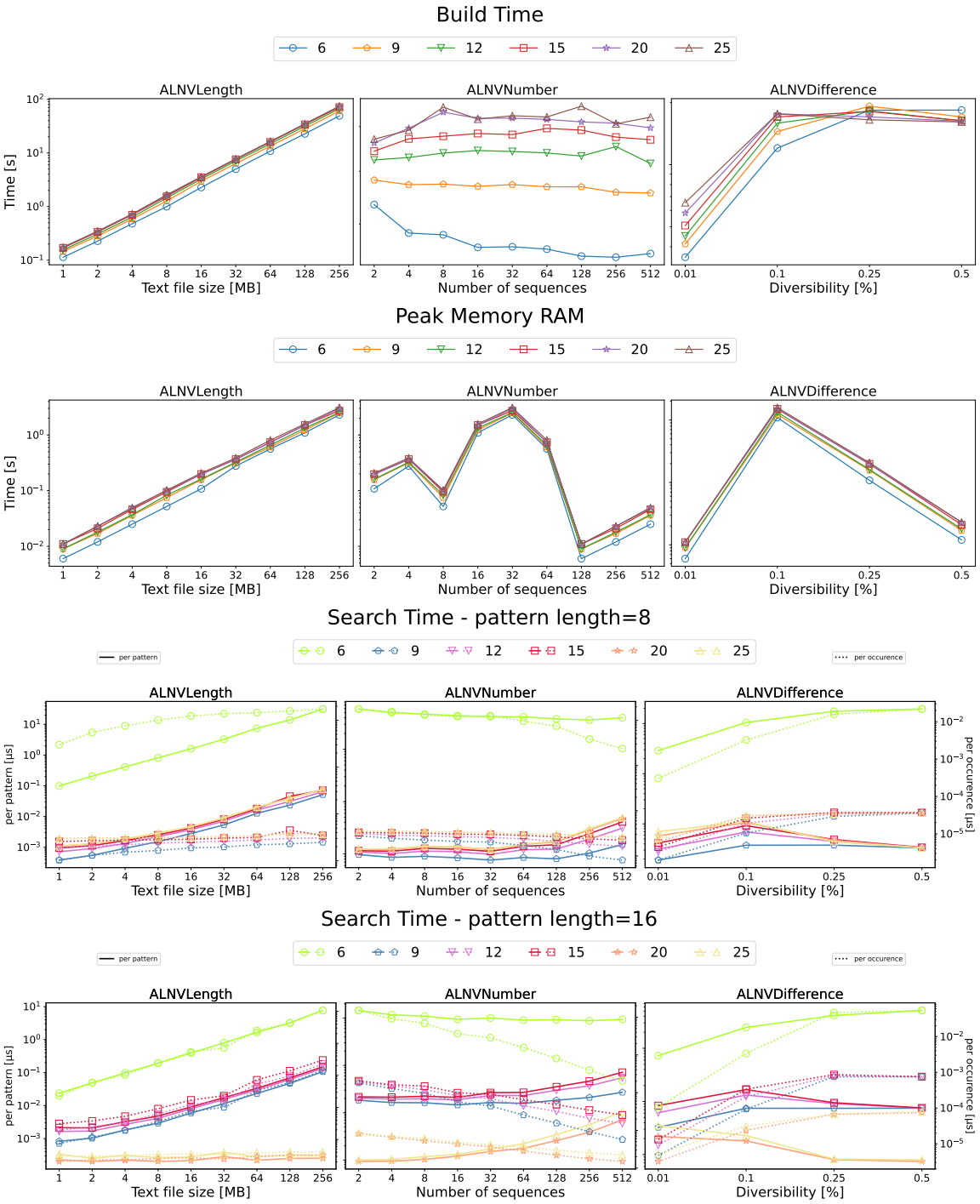


Figure 4.4: Context influence on construction and search time

Chapter 5

Result discussion

5.1 Suitable setting of BIO-FMI

Figure 4.1 shows that the ratio of pattern length to input size has not any influence on pattern frequency, hence we can talk about suitable patterns generally and predict successful usage of the algorithm in a reasonable time.

The biggest fall can be observed for patterns shorter than the value of 10. Another moderate decline set in patterns of lengths of about three hundred, where the patterns are usually unique in texts. As the result, it depends on user preferences. In general, any pattern shorter than ten should be avoided for algorithm BIO-FMI and this value also applies to stored context length.

5.2 BIO-FMI efficiency discussion

Unlike the others, the BIO-FMI gives a positive opinion on the construction part. In the search part, under certain conditions, implementation can be a good tool and surpass algorithms of LZ-RLBWT and r-index.

The implementation of BIO-FMI is very quick in its construction and the results respond to the theoretical estimation of linear time. The results in Figure 4.2 confirm our assertion. The length of sequences, or general input text files, has a dramatic effect on construction time. This kind of behavior is expected and it does not differ from other algorithms. It can be observed, that the number of sequences has a nearly small effect on construction time. This can be used for searching in short reads in sequence analyses. We do not observe any unexpected results even on the last type of the dataset where the time consumption increases with an interesting sequence difference.

Algorithm search time has unpleasant results in comparison with the LZ-RLBWT and r-index with regard to the input size. As you can see Figure 4.3, the BIO-FMI upward trend is faster than the others. We can assume, that our index usage will be limited by this parameter. The performance on a variable number of sequences is an inappropriate result for discussion because with the

increasing number, the length of sequences is shorter and so the comparison is quicker.

The influence of context length copies the same trends for all datasets and parameters in the building part. With the increasing context both the time and memory increase themselves. Notice, that the differences are smaller with the increasing context, so from some point are unobservable.

During the search part, with a lower context length, the search time increases. This observation is in the line with the theory of algorithm, where the more information and occurrences there are in the non-reference index, the more time is saved.

5.3 BIO-FMI over EDS discussion

EDS results give us a positive opinion on smaller text sizes. At present, there is no comparison with another index available, but the comparison with its original version over *.aln* gives satisfying results.

At this time, the problematic part is input reading and parsing into the required form, which takes enormous amount of memory and thus prolongs the construction time.

Context length follows the same trends as in the ALN version, but during the pattern matching the suitable small context, length shows better results.

5.4 Summary

To summarize, searching in BIO-FMI is expensive with increasing the text size. This fact leads to limited usage of algorithms over bigger files. There are a few exceptions but overall with increasing the context length the efficiency seems to be higher and takes some additional memory.

The patterns and context length parameter should have a length of at least 10. At the same time, for every context length, the pattern should be either a multiple of the context or one less to the best distribution of chunk lengths.

Implementation over EDS gives very good results against implementation over ALN, but it takes a lot of time and memory to read the input file. The EDS implementation has a strong dependency on the setting of *context_length* and the length of seeds. This approach can be useful for very high repetitive texts with lower stored context.

Conclusion

To conclude this document, the survey of indexing in genomes was finished including a basic introduction to sequence analysis and variant storing. The current indexing way was presented and shortcomings were explained.

Furthermore, the thesis describes the algorithm BIO-FMI, the BIO-FMI application was written to test the ability of the algorithm to adapt itself to different types of data. For the first time, the full version including indel changes recognizing was implemented and tested. Moreover, the document introduces the new adaptation of BIO-FMI on elastic degenerate strings and gives us a new approach to indexing this structure adequate for pan-genomic representation.

The results show that the algorithm is highly dependent on the choice of the stored context length for a specific pattern length, and its value cannot be constant. On the other hand, this parameter can be deduced for groups of problems and implemented in potential later versions of the BIO-FMI tool. The same conclusions apply to the length of the pattern.

Implementation over EDS provides good results and confirms its usability in this field.

Future work

In the future, there is much work to optimize the implementation. An improper approach was implemented for reading input text in EDS format, which takes a lot of time and more, its size enormously grows with increasing stored context length.

My type of implementation takes a lot of RAM memory, which can be probably optimized with the creation of metafiles during the process. After reading optimization, the implementation over the EDS string and the structure of the result can be improved by a new, appropriate type of hash table, which gives us a constant time of validation and improves the search time.

During the experiment testing, some kinds of pattern mismatching were detected as a consequence of a high number of special cases. Even if their existence does not cause great harm to experiments because of the huge range of experiments, they need to be detected and resolved.

Implementation over elastic degenerate strings needs to be compared with other algorithms, As one suitable algorithm seems to be SOPanG2.

Finally, over some time, I would like to extend a range of experiments on real datasets such as 1,000 human genomes. For this purpose, there needs to be some way of extracting the required information from VCF files. The genome of eukaryotes contains more than one chromosome, so the chromosome selection and distinct pattern matching over them could be implemented.

Bibliography

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J Mol Biol*, 1990. PMID: 2231712.

[2] David G. Andersen. A simple introduction to compressed suffix arrays. 2012.

[3] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Combinatorial Pattern Matching*, volume 9133, 02 2015.

[4] Rene Rivera Beman Dawes, David Abrahams, 2005. Version 1.0.

[5] Bonnie Berger, Michael S. Waterman, and Yun William Yu. Levenshtein distance, sequence comparison and biological database search. *IEEE Transactions on Information Theory*, 67(6):3287–3294, 2021.

[6] James K. Bonfield, John Marshall, Petr Danecek, Heng Li, Valeriu Ohan, Andrew Whitwham, Thomas Keane, and Robert M. Davies. Htslib: C library for reading/writing high-throughput sequencing data. *GigaScience*, 10(2):giab007, Feb 2021. 33594436.

[7] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Carnegie Mellon University, 1994.

[8] Aleksander Cisłak and Szymon Grabowski. SOPanG 2: online searching over a pan-genome without false positives, 2020.

[9] Clark, David. *Compact PAT trees.* PhD thesis, University of Waterloo, 1997.

[10] Laura Clarke, Susan Fairley, Xiangqun Zheng-Bradley, Ian Streeter, Emily Perry, Ernesto Lowy, Anne-Marie Tassé, and Paul Flicek. The

international Genome sample resource (IGSR): A worldwide collection of genome variation incorporating the 1000 Genomes Project data. *Nucleic Acids Research*, 45(D1):D854–D859, 09 2016.

[11] P J Cock, C J Fields, N Goto, M L Heuer, and P M Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic Acids Res*, 38(6):1767–1771, April 2010.

[12] NCBI Resource Coordinators. Database resources of the national center for biotechnology information. *Nucleic acids research*, 44(D1):D7–D19, Jan 2016. 26615191.

[13] Draesslerová Dominika. Bioinformatics index tool for elastic degenerate string matching, 2022. GitHub.

[14] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. *Proc FOCS 2000*, 2000:390–398, 02 2000.

[15] Philippe Fournier-Viger, Tin Truong Chi, Youxi Wu, Jun-Feng Qu, Jerry Chun-Wei Lin, and Zhitian Li. *Finding Periodic Patterns in Multiple Sequences*, pages 81–103. Springer Singapore, Singapore, 2021.

[16] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space, 2017.

[17] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67:1–54, 01 2020.

[18] Erik Garrison, Jouni Sirén, Adam M. Novak, Glenn Hickey, Jordan M. Eizenga, Eric T. Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F. Lin, Benedict Paten, and Richard Durbin. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology*, 36(9):875–879, Oct 2018.

[19] Simon Gog. Sdsl - succinct data structure library, 2019. GitHub.

[20] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[21] Szymon Grabowski and Marcin Raniszewski. Sampling the suffix array with minimizers. In Costas Iliopoulos, Simon Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval*, pages 287–298, Cham, 2015. Springer International Publishing.

[22] Jan Holub. Lecture notes of efficient text pattern matching, 2021. Czech Technical University in Prague (CTU).

[23] Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate strings. *CoRR*, abs/1610.08111, 2016.

[24] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[25] Juha Kärkkäinen and Esko Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching (extended abstract). In *Proc. 3rd South American Workshop on String Processing (WSP'96*, pages 141–155. Carleton University Press, 1996.

[26] Ben Langmead. Lecture notes Burrows-Wheeler Transform and FM Index. Johns Hopkins University.

[27] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 06 2009.

[28] E. Lieberzeitova. Dna sequence type classification using deep learning. Master's thesis, University of Chemical Technology in Prague (UCT), 2022.

[29] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, oct 1993.

[30] J. Ian Munro. Tables. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[31] U.S. National Library of Medicine National Center for Biotechnology Information. Fasta format for nucleotide sequences, 2021.

[32] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39:2, 2007.

[33] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[34] Danecek P, Auton A, Goncalo Abecasis, Albers CA, Banks E, DePristo MA, Handsaker RE, Lunter G, Stephen Sherry, McVean G, Genomes R, Altshuler D, Bentley D, Chakravarti A, Clark A, De F, Donnelly P, and Ren Z. The variant call format and VCFtools. *Bioinformatics*, 27:2156–8, 08 2011.

[35] Prashant Pandey, Yinjie Gao, and Carl Kingsford. VariantStore: an index for large-scale genomic variant search. *Genome biology*, 22(1):231–231, Aug 2021. 34412679.

[36] Gonzalo Navarro Paolo Ferragina. Pizza&chili corpus compressed indexes and their testbeds, 2005.

[37] Jonathan Pevsner. *Bioinformatics and functional genomics*. Wiley-Blackwell, Chichester, third edition, 2015.

[38] Nicola Prezza. lz-rlbwt: Run-length compressed burrows-wheeler transform with lz77 suffix array sampling, 2022. GitHub.

[39] Nicola Prezza. r-index: the run-length bwt index, 2022. GitHub.

[40] Petr Procházka and Jan Holub. Compressing similar biological sequences using fm-index. In *2014 Data Compression Conference*, pages 312–321, 2014.

[41] S. Ratnasingham and P.D.N. Hebert. Bold: The barcode of life data system. *Molecular Ecology Notes*, 7:355–364, 2007.

[42] Jason A. Reuter, Damek V. Spacek, and Michael P. Snyder. High-throughput sequencing technologies. *Molecular cell*, 58(4):586–597, May 2015. 26000844.

[43] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

Appendix A

Experiment details

A.1 ALN Build

A.1.1 Construction time

Table A.1: Construction time - ALNVLength

	LZ-RLBWT	r-index	BIO-FMI
1	86.0	0.41670	0.11160
2	172.0	0.82653	0.22384
4	340.0	1.75622	0.45388
8	671.0	4.12301	0.97114
16	1342.0	9.81696	2.23030
32	2691.0	22.01270	4.87541
64	5483.0	47.71760	10.82420
128	10929.0	99.66440	22.64540
256	22396.0	210.83900	48.83330

Table A.2: Construction time - ALNVNumber

	LZ-RLBWT	r-index	BIO-FMI
2	412.0	1.97260	0.53477
4	387.0	1.94424	0.48480
8	371.0	1.85612	0.48184
16	362.0	1.82452	0.46138
32	357.0	1.83036	0.46233
64	346.0	1.78487	0.45857
128	332.0	1.74929	0.44765
256	320.0	1.71574	0.44589
512	314.0	1.76860	0.45159

Table A.3: Construction time - ALNVDifference

	LZ-RLBWT	r-index	BIO-FMI
0.01	43.0	0.33983	0.03558
0.1	87.0	0.40988	0.12028
0.25	114.0	0.46160	0.18361
0.5	122.0	0.47724	0.18382

A.1.2 Peak memory RAM usage

Table A.4: Peak memory - ALNVLength

	LZ-RLBWT	r-index	BIO-FMI
1	23.4688	26.464	0.015
2	32.2930	52.580	0.028
4	49.5703	104.988	0.050
8	83.4570	210.524	0.095
16	155.6170	401.704	0.204
32	293.6880	810.776	0.391
64	577.4410	1666.092	0.758
128	1054.8400	3269.280	1.501
256	2039.0100	6908.016	3.046

Table A.5: Peak memory - ALNVNumber

	LZ-RLBWT	r-index	BIO-FMI
2	53.7734	162.132	0.204
4	51.8672	132.724	0.391
8	51.4102	124.428	0.095
16	50.5234	120.012	1.501
32	50.5117	115.872	3.046
64	49.6250	108.268	0.758
128	48.1953	99.664	0.015
256	47.7891	91.024	0.028
512	47.3711	87.016	0.050

Table A.6: Peak memory - ALNVDifference

	LZ-RLBWT	r-index	BIO-FMI
0.01	20.3398	13.280	0.015
0.1	23.4961	26.680	1.501
0.25	25.4844	43.744	0.204
0.5	25.8438	48.696	0.028

A.1.3 Compression ratio

Table A.7: Compression ratio - ALNVLength

	LZ-RLBWT	r-index	BIO-FMI
1	1193.23862	2323.72040	993.81148
2	1529.16961	3242.42933	1266.70789
4	790.01924	1786.37073	662.05125
8	10.22369	24.33645	8.62447
16	1598.21366	3986.19785	1344.14156
32	59812.65997	155875.67719	49932.50646
64	95680.38908	262826.50948	76713.01003
128	129456.55008	371840.04314	108073.35901
256	105917.39357	321146.25046	83290.62917

Table A.8: Compression ratio - ALNVNumber

	LZ-RLBWT	r-index	BIO-FMI
2	522.02372	1370.56856	233.28269
4	8851.49213	22444.55222	5179.27611
8	1.46282	3.58447	1.00218
16	4364.39000	10544.37769	3224.41615
32	1673.60758	3992.32000	1290.07152
64	5822.86986	13454.53726	4834.75417
128	4500.37727	9893.93727	3962.76455
256	2725.63706	5660.92412	2573.19118
512	673.98508	1400.90246	663.85123

Table A.9: Compression ratio - ALNVDifference

	LZ-RLBWT	r-index	BIO-FMI
0.01	423.02283	785.22283	190.9324
0.1	1029.08571	2024.80000	832.6772
0.25	143.91106	338.40377	117.1601
0.5	1202.43835	2958.55398	687.4006

A.2 ALN Search

Tables on the left contain search time per occurrence, on the right per pattern.

Table A.10: Search time per occurence - ALNVLength 8

	LZ-RLBWT	r-index	BIO-FMI
1	0.00682	0.00026	0.00001
2	0.00771	0.00045	0.00001
4	0.00881	0.00048	0.00002
8	0.00934	0.00042	0.00002
16	0.01137	0.00060	0.00002
32	0.01288	0.00043	0.00002
64	0.01467	0.00044	0.00000
128	0.01560	0.00051	0.00000
256	0.01694	0.00056	0.00000

Table A.11: Search time per pattern - ALNVLength 8

	LZ-RLBWT	r-index	BIO-FMI
1	0.26	0.01	0.000382
2	0.34	0.02	0.000543
4	0.55	0.03	0.000923
8	0.88	0.04	0.001515
16	1.72	0.09	0.002792
32	3.58	0.12	0.005333
64	8.73	0.26	0.000128
128	16.40	0.54	0.000224
256	35.86	1.18	0.000508

Table A.12: Search time per occurence - ALNVNumber 8

	LZ-RLBWT	r-index	BIO-FMI
2	0.01273	0.00085	0.00003
4	0.01144	0.00056	0.00002
8	0.01097	0.00073	0.00002
16	0.01021	0.00071	0.00002
32	0.01037	0.00078	0.00002
64	0.00917	0.00059	0.00002
128	0.00807	0.00050	0.00001
256	0.00671	0.00037	0.00001
512	0.00584	0.00026	0.00001

Table A.13: Search time per pattern - ALNVNumber 8

	LZ-RLBWT	r-index	BIO-FMI
2	0.45	0.03	0.000972
4	0.41	0.02	0.000860
8	0.45	0.03	0.000902
16	0.43	0.03	0.000844
32	0.40	0.03	0.000764
64	0.47	0.03	0.000855
128	0.48	0.03	0.000808
256	0.73	0.04	0.001039
512	1.13	0.05	0.001515

Table A.14: Search time per occurence - ALNVDifference 8

	LZ-RLBWT	r-index	BIO-FMI
0.01	0.00239	0.00021	0.00000
0.1	0.00702	0.00027	0.00001
0.25	0.01149	0.00077	0.00003
0.5	0.01303	0.00109	0.00004

Table A.15: Search time per pattern - ALNVDifference 8

	LZ-RLBWT	r-index	BIO-FMI
0.01	0.23	0.02	0.00019
0.1	0.26	0.01	0.00038
0.25	0.15	0.01	0.00038
0.5	0.12	0.01	0.00033

Table A.16: Search time per occurence - ALNVLength 16

	LZ-RLBWT	r-index	BIO-FMI
1	0.01490	0.00186	0.00008
2	0.01655	0.00236	0.00012
4	0.01749	0.00309	0.00019
8	0.01993	0.00374	0.00036
16	0.02290	0.00509	0.00075
32	0.02111	0.00352	0.00100
64	0.02766	0.00553	0.00003
128	0.02760	0.00627	0.00006
256	0.03256	0.00678	0.00014

Table A.17: Search time per pattern - ALNVLength 16

	LZ-RLBWT	r-index	BIO-FMI
1	0.16	0.02	0.000830
2	0.14	0.02	0.001026
4	0.17	0.03	0.001831
8	0.16	0.03	0.002928
16	0.18	0.04	0.005937
32	0.24	0.04	0.011560
64	0.20	0.04	0.000234
128	0.22	0.05	0.000481
256	0.24	0.05	0.001069

Table A.18: Search time per occurence - ALNVNumber 16

	LZ-RLBWT	r-index	BIO-FMI
2	0.04237	0.02542	0.00182
4	0.03311	0.01324	0.00128
8	0.03191	0.01064	0.00097
16	0.02482	0.00709	0.00057
32	0.02163	0.00721	0.00044
64	0.01923	0.00321	0.00026
128	0.01682	0.00252	0.00016
256	0.01408	0.00136	0.00009
512	0.01288	0.00066	0.00006

Table A.19: Search time per pattern - ALNVNumber 16

	LZ-RLBWT	r-index	BIO-FMI
2	0.05	0.03	0.001835
4	0.05	0.02	0.001639
8	0.06	0.02	0.001623
16	0.07	0.02	0.001459
32	0.09	0.03	0.001646
64	0.12	0.02	0.001575
128	0.20	0.03	0.001835
256	0.31	0.03	0.002098
512	0.59	0.03	0.002699

Table A.20: Search time per occurence - ALNVDifference 16

	LZ-RLBWT	r-index	BIO-FMI
0.01	0.00349	0.00029	0.00000
0.1	0.01567	0.00131	0.00009
0.25	0.03333	0.00833	0.00075
0.5	0.04000	0.01000	0.00075

Table A.21: Search time per pattern - ALNVDifference 16

	LZ-RLBWT	r-index	BIO-FMI
0.01	0.24	0.02	0.000325
0.1	0.12	0.01	0.000744
0.25	0.04	0.01	0.000743
0.5	0.04	0.01	0.000747

A.3 EDS Build

A.3.1 Construction time

Table A.22: Construction time - EDSVLength

	6	9	12	15
01	0.03836	0.05585	0.13257	0.25527
1	0.29521	0.63138	1.21961	2.23245
2	0.61274	1.15504	2.86576	5.56061
4	1.28601	2.52741	7.15745	82.14360
8	3.05744	5.29240	13.09260	23.94830

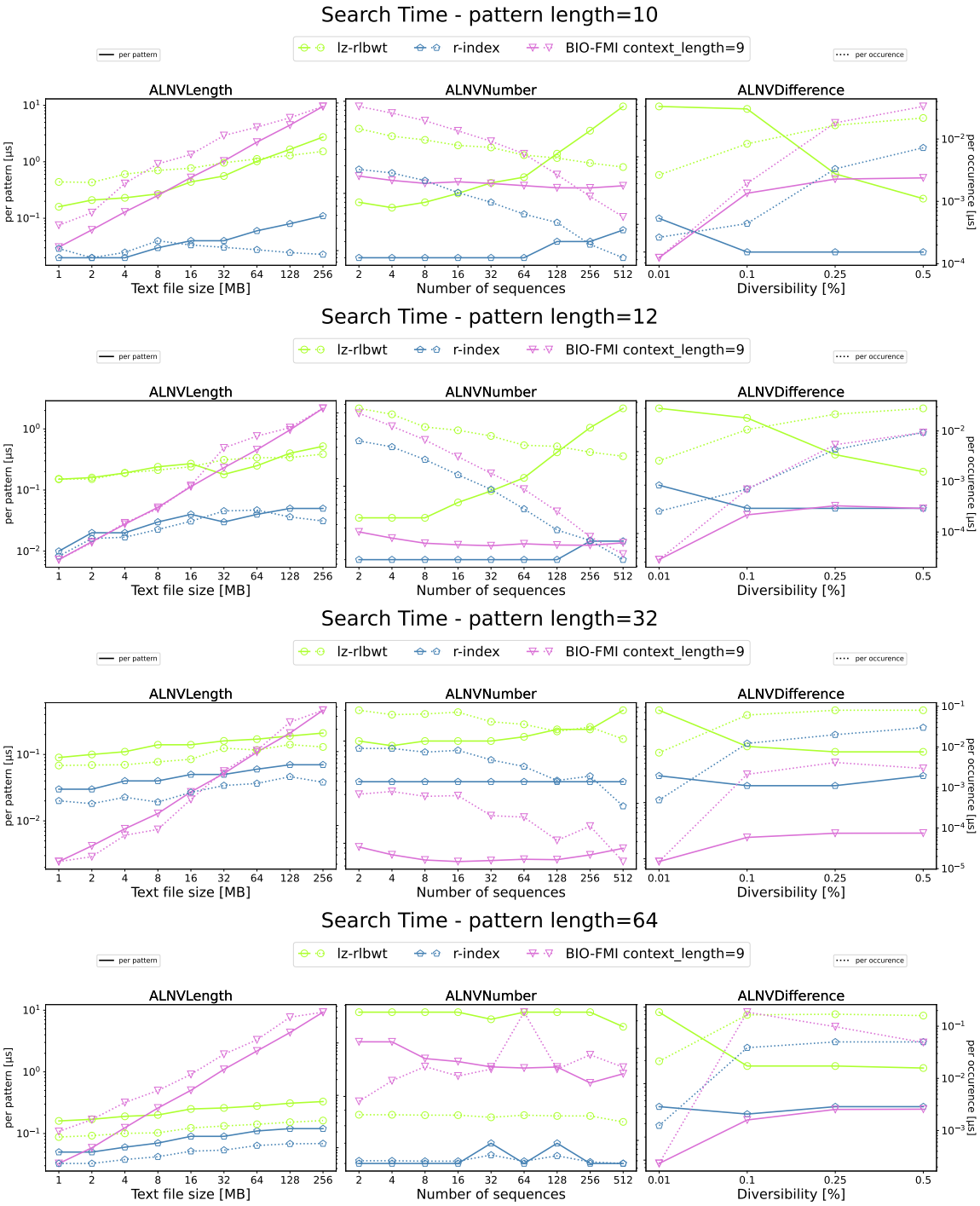


Figure A.1: Search result for other pattern lengths - ALN

Table A.23: Construction time - EDSVNumber

	6	9	12	15
01	0.32616	0.33551	0.34344	0.35066
05	0.19944	0.23574	0.28136	0.36482
1	0.60954	1.55617	13.26390	23.20900

A.3.2 Peak memory RAM usage

Table A.24: Peak memory - EDSVLength

	6	9	12	15
01	0.003	0.004	0.010	0.018
1	0.011	0.040	0.091	0.170
2	0.022	0.083	0.153	0.344
4	0.052	0.144	0.532	4.871
8	0.138	0.308	0.715	1.497

Table A.25: Peak memory - EDSVNumber

	6	9	12	15
01	0.017	0.019	0.019	0.019
05	0.009	0.008	0.012	0.020
1	0.023	0.096	0.715	1.351

A.3.3 Compression ratio

Table A.26: Compression ratio - EDSVLength

	6	9	12	15
01	309.36149	309.36149	309.36149	309.36149
1	2470.78324	2470.78324	2470.78324	2470.78324
2	503.68496	503.68496	503.68496	503.68496
4	7696.60964	7696.60964	7696.60964	7696.60964
8	8572.92093	8572.92093	8572.92093	8572.92093

Table A.27: Compression ratio - EDSVNumber

	6	9	12	15
01	2883.92898	2883.92898	2883.92898	2883.92898
05	1595.58121	1595.58121	1595.58121	1595.58121
1	505.24646	505.24646	505.24646	505.24646

A.4 EDS Search

The first rows contain search time per pattern, and second per occurrence.

Table A.28: Search time per pattern - EDSVLength 8

	6	9	12	15
01	0.000388	0.000004	0.000008	0.000009
1	0.008622	0.000011	0.000016	0.000024
2	0.025999	0.000013	0.000019	0.000029
4	0.094414	0.000027	0.000042	0.000062
8	0.363981	0.000053	0.000084	0.000204

Table A.29: Search time per occurence - EDSVLength 8

	6	9	12	15
01	0.000154	2.551880e-07	1.452880e-07	1.448900e-07
1	0.000433	2.388530e-07	1.854960e-07	1.628110e-07
2	0.000700	2.667310e-07	2.339370e-07	1.602150e-07
4	0.000914	2.826510e-07	2.147810e-07	1.650940e-07
8	0.001580	2.792700e-07	1.734850e-07	8.893520e-08

Table A.30: Search time per pattern - EDSVNumber 8

	6	9	12	15
01	0.010712	0.000008	0.000008	0.000008
1	0.024716	0.000018	0.000034	0.000093
05	0.004560	0.000005	0.000006	0.000007

Table A.31: Search time per occurence - EDSVNumber 8

	6	9	12	15
01	0.000505	3.533950e-07	3.407910e-07	3.303850e-07
1	0.000501	2.078200e-07	1.567230e-07	9.330300e-08
05	0.000301	3.393440e-07	3.210950e-07	2.692960e-07

Table A.32: Search time per pattern - EDSVLength 16

	6	9	12	15
01	0.000082	0.000004	0.000006	0.000008
1	0.000806	0.000020	0.000031	0.000042
2	0.001859	0.000030	0.000047	0.000216
4	0.004295	0.000058	0.000111	0.000655
8	0.011260	0.000126	0.000230	0.000416

Table A.33: Search time per occurence - EDSVLength 16

	6	9	12	15
01	0.000175	0.000008	0.000014	0.000017
1	0.001831	0.000046	0.000071	0.000096
2	0.004647	0.000074	0.000119	0.000541
4	0.008766	0.000119	0.000226	0.001336
8	0.030431	0.000340	0.000621	0.001124

Table A.34: Search time per pattern - EDSVNumber 16

	6	9	12	15
01	0.001065	0.000013	0.000013	0.000014
1	0.001794	0.000031	0.000050	0.000118
05	0.000620	0.000010	0.000011	0.000013

Table A.35: Search time per occurence - EDSVNumber 16

	6	9	12	15
01	0.001972	0.000024	0.000025	0.000026
1	0.004376	0.000074	0.000121	0.000287
05	0.001266	0.000020	0.000023	0.000026

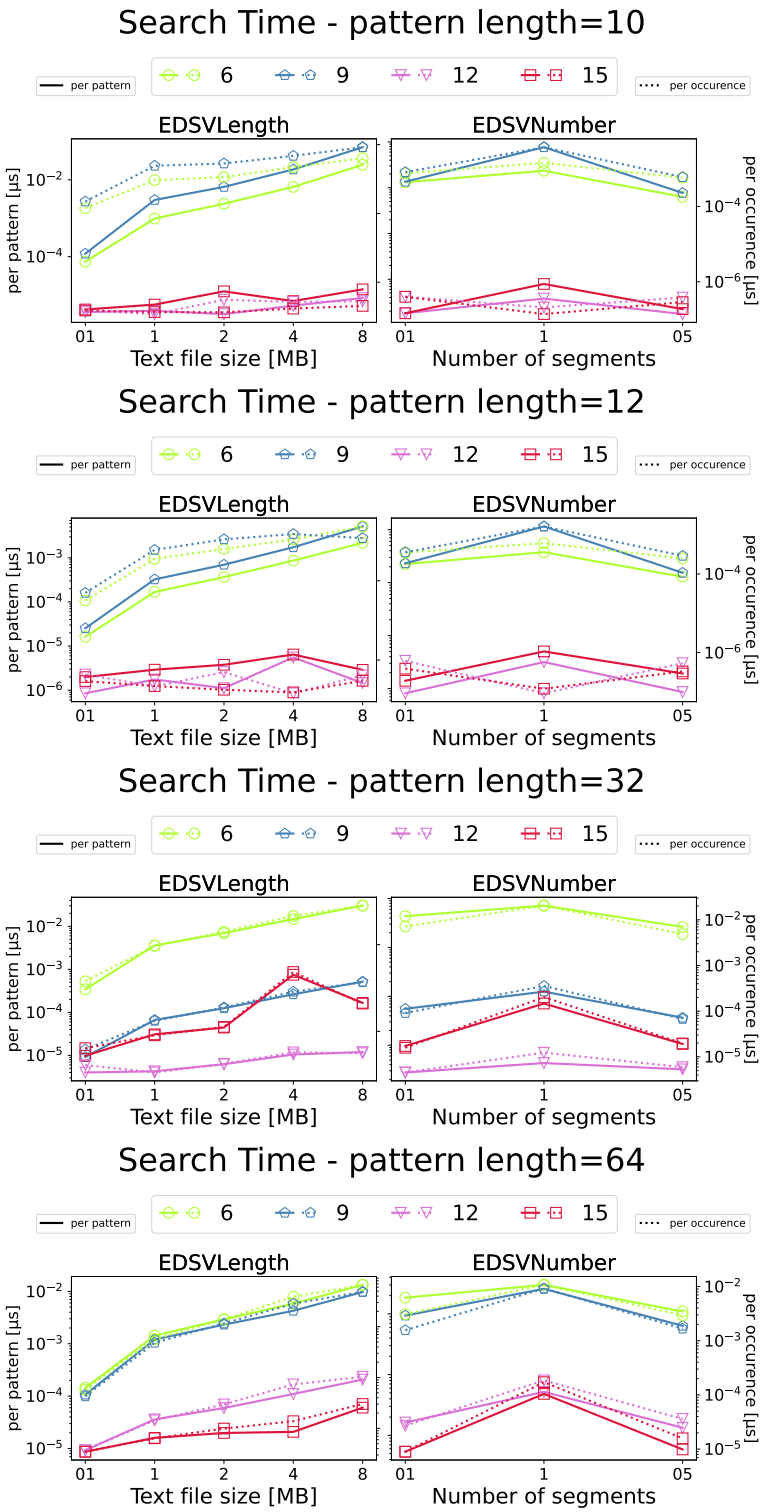


Figure A.2: Search result for other pattern lengths - EDS

Appendix B

Usage

B.1 Installation

First make sure, that the SDSL library is installed on your device. If not, see the installation manual [19]. Second, make sure, that you have the required standard of C++ for this application, which is C++17. Clone BIO-FMI from GitHub repository [13] and install it with the following commands:

```
$   mkdir build
$   cd build
$   cmake ..
$   make
```

The applications *bio-fmi-build* and *bio-fmi-locate* are located in build folder after successful compilation.

B.2 Usage

```
$ ./bio-fmi-build --help-verbose
This software is called BIO-FMI (Index for set of genomes and pangenomes).
It can be used for pattern matching in elastic-degenerate (ED text).
Authors: Petr Prochazka, Jan Holub.
Input format can be saved in .aln (ALN) file - alignment of sequences,
every sequence in new line; or in .eds file - {A,C,}GAAT{,A,AT}ATT,
where the strings in bracket are ordered ascending.BIO-FMI
return start positions of pattern occurences.


Usage: ./bio-fmi-build [options] <input_file_name>

Allowed options:
```

```
  --help                        produce help message
  --help-verbose                display verbose help message
  -v [ --version ]              display version info
  -s [ --silent ]               silent mode
  -r [ --repetition ] arg       number of repetition - for experiment needs
  -l [ --context_length ] arg length of chunk and stored context, default 5
  -o [ --basefolder ] arg         use <basefolder> as prefix for all index files.
                                Default: current folder is the specified
                                input_file_name
  -i [ --input-file ] arg       input file

Input text file (positional parameter 1 or named parameter -i or
--in-text-file) should contain the file with extension .aln in the
format of sequence -AC--GT-CGTA and every sequence in new line or
.eds in the format {A,C,}GAAT{,A,AT}ATT.
Context length (optional named parameter -l or --context_length
with default value of 6) divide input pattern into chunks of this
length and during the construction save same length right and left
context of text input.

$ ./bio-fmi-locate --help-verbose
This software is called BIO-FMI (Index for set of genomes and pangenomes).
It can be used for pattern matching in elastic-degenerate (ED text).
Authors: Petr Prochazka, Jan Holub.
Input format can be saved in .aln (ALN) file - alignment of sequences,
every sequence in new line; or in .eds file - {A,C,}GAAT{,A,AT}ATT,
where the strings in bracket are ordered ascending.
BIO-FMI return start positions of pattern occurences.

Usage: ./bio-fmi-locate [options] <index_basename> <pattern_file>

Parameters:
  -h [ --help ]             display help message
  --help-verbose            display verbose help message
  -v [ --version ]          display version info
  -s [ --silent ]           silent mode
  -p [ --pattern ]          print occurences of every pattern
  -i [ --index-path ] arg   input text file path (positional arg 1)
  -I [ --pattern-file ] arg input pattern file path (positional arg 2)

Input text file (positional parameter 1 or named parameter -i or
--in-text-file) should contain the file with extension .aln in
the format of sequence -AC--GT-CGTA and every sequence in new line or
.eds in the format {A,C,}GAAT{,A,AT}ATT.
```

```
Input pattern file (positional parameter 2 or named parameter -I or
--in-pattern-file) should contain the information about number an
length of patterns followed by one line of concatenated patterns.
```

Every application supports silent mode with `-s` parameter. Within this the all additive structures are printed. Parameter `-p` is used to print only number of occurrences for every pattern. Parameter `-r` can be used for more accurately measurement of build time, this parameter repeats index construction r-times and print average time.

Appendix C

Acronyms

LZ-RLBWT Lempel Ziv Run length Burrows Wheeler transformation

SDSL Succinct Data Structure Library

DNA Deoxyribonucleic acid

EDS Elastic degenerate string

SA Suffix array

BWT Burrows Wheeler transformation

WT Wavelet tree

mRNA messenger ribonucleic acid

HTS High-Throughput sequencing

NGS New generation sequencing

Appendix D

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `bio-fmi` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format